Type Inference on Executables

JUAN CABALLERO, IMDEA Software Institute
ZHIQIANG LIN, University of Texas at Dallas

In many applications, source code and debugging symbols of a target program are not available, and the only thing that we can access is the program executable. A fundamental challenge with executables is that, during compilation, critical information such as variables and types is lost. Given that typed variables provide fundamental semantics of a program, for the last 16 years, a large amount of research has been carried out on binary code type inference, a challenging task that aims to infer typed variables from executables (also referred to as binary code). In this article, we systematize the area of binary code type inference according to its most important dimensions: the applications that motivate its importance, the approaches used, the types that those approaches infer, the implementation of those approaches, and how the inference results are evaluated. We also discuss limitations, underdeveloped problems and open challenges, and propose further applications.

Categories and Subject Descriptors: D.3.3 [**Language Constructs and Features**]: Data types and structures; D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Languages, Security

Additional Key Words and Phrases: Type inference, program executables, binary code analysis

ACM Reference Format:
Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. ACM Comput. Surv. 48, 4, Article 65 (May 2016), 35 pages.
DOI: http://dx.doi.org/10.1145/2896499

1. INTRODUCTION

Being the final deliverable of software, executables (or binary code, as we use both terms interchangeably) are everywhere. They contain the final code that runs on a system and truly represent the program behavior. In many situations, such as when analyzing commercial-off-the-shelf (COTS) programs, malware, or legacy programs, we can access program executables only since the source code and debugging symbols are not available.

Analyzing executables is challenging because, during compilation, much program information is lost. One particularly critical piece of missing information is the *variables* that store the data, and their *type*, which constrains how the data is stored, manipulated, and interpreted. Given their importance, a large amount of research has been

This research was partially supported by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731, by the Spanish Government through the StrongSoft Grant TIN2012-39391-C04-01, and by the U.S Government through NSF Grant 1453011 and AFOSR Grant FA9550-14-1-0119. All opinions, findings and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.
Authors' addresses: J. Caballero, Instituto IMDEA Software, Campus de Montegancedo S/N, 28223 Pozuelo de Alarcon, Madrid, Spain; email: Juan.caballero@imdea.org; Z. Lin, Computer Science Department, University of Texas at Dallas, 800 W. Campbell Rd, Richardson, TX 75080, USA; email: zhiqiang.lin@utdallas.edu.

carried out over the last 16 years on *binary code type inference*, a challenging task that aims to infer typed variables from executables.

In this article, we systematize the area of binary code type inference according to its most important dimensions: the applications that motivate its importance, the proposed approaches, the types that those approaches infer, the implementation of those approaches, and how the inference results are evaluated. We also discuss limitations, underdeveloped problems, and open challenges.

Binary code type inference is required for, or significantly benefits, many applications such as decompilation [Cifuentes 1994; Breuer and Bowen 1994; Schwartz et al. 2013; Zeng et al. 2013], binary code rewriting [Sites et al. 1993; Larus and Ball 1994; Schwarz et al. 2001; Abadi et al. 2009], vulnerability detection and analysis [Slowinska et al. 2011; Caballero et al. 2012a], binary code reuse [Caballero et al. 2010; Kolbitsch et al. 2010], protocol reverse engineering [Caballero et al. 2007; Wondracek et al. 2008; Lin et al. 2008; Cui et al. 2008], virtual machine introspection [Payne et al. 2008; Dolan-Gavitt et al. 2011; Vogl et al. 2014], game hacking [Bursztein et al. 2011; Urbina et al. 2014], hooking [Yin et al. 2008; Payne et al. 2008; Vogl et al. 2014], and malware analysis [Jiang et al. 2007; Cozzie et al. 2008].

We have systematically compared 38 binary code type inference works and have examined over 100 papers in this area. Our analysis shows that existing binary code type inference solutions widely differ in their approaches. For example, they can use static analysis, dynamic analysis, and combinations of them. They can infer types using value-based or flow-based inference starting from different sources of type information. Our analysis also reveals that they widely differ in the types that they infer, which include primitive types such as integers, floats, and pointers; aggregate data structures such as records and arrays; classes in object-oriented programs; and higher-level recursive types such as lists and trees. Even program code can be typed and types can be recovered for function prototypes. However, no single work tries to infer all types and most solutions focus on a small set of types. We also show that the field of binary code type inference is multidisciplinary; works have been published in a variety of areas including security, systems, software engineering, and programming languages. Given the variety of approaches, types inferred, and the interdisciplinary character, it becomes critical to have an article that jointly discusses them.

We also examine existing solutions according to their implementation, including the platforms they built on, the intermediate representations used, and the architectures and operating systems supported. Furthermore, we systematize the evaluation of the type inference results, examining the benchmarks and methodologies used. We observe that many works perform qualitative evaluation for a specific application and highlight the need for quantified results. We examine the accuracy metrics that have been proposed and whether works compare with prior solutions. Our analysis suggests the need for a common type representation for easier result comparison.

An overview of the organization of this article is presented in Figure 1. We begin in Section 2 with an overview of binary code type inference and the scope of the article. In Section 3, we describe the applications that motivate binary code type inference. In Section 4, we systematize binary code type inference works according to their approaches; in Section 5, we systematize the inferred types. We discuss how each system gets implemented and evaluated in Section 6 and Section 7, respectively. In Section 8, we discuss other insights beyond the ones in Section 2 through Section 7, point out open areas of research, and introduce future trends on binary code type inference. We present our conclusions in Section 9.

2. OVERVIEW

In this section, we discuss the goal and scope of binary code type inference.

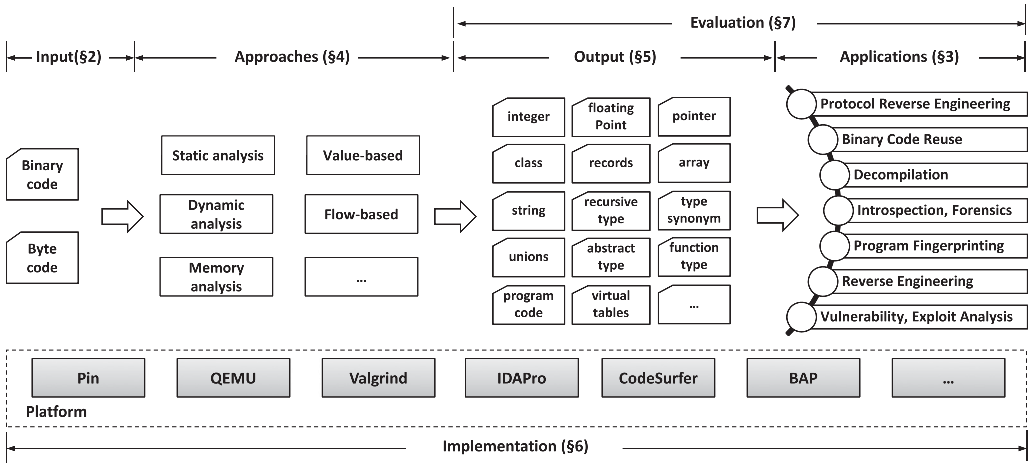


Fig. 1. Article organization and overview.

2.1. Binary Code Type Inference

The input to type inference can be source code, byte code, or binary code. For example, the field of type inference started from programming languages such as ML, for which developers do not specify types in the source code. Instead, types are inferred directly from the source code during compilation [Milner 1978]. This article systematizes works on binary code type inference, that is, those that recover types from compiled programs. Note that compiled programs are distributed as binary code, which is challenging since during compilation variable and type information is not included in the resulting executable. In contrast, interpreted code is distributed as source code or byte code with much type information.

Binary code type inference is affected by the programming language, compiler, operating system, and target architecture. The programming language defines built-in types and mechanisms for the programmer to define one's own types. The compiler chooses the application binary interface (ABI), which covers type representations, data structure alignment, calling conventions, layout of classes with inheritance, the object file format (e.g., PE/ELF), and so on. The target architecture also influences data representation, for example, big-endian vs. little-endian and integer/pointer size. Even the operating system may affect the representation, for example, Windows may convert strings internally to UTF-16.

While the proposed approaches may be generic, most are evaluated on specific combinations of programming languages and architectures. As shown in Section 6, the most common target platform for binary code type inference is x86 (36/38 approaches evaluated), followed by x86-64 (7 approaches). All approaches evaluate at least on C/C++ programs.

2.2. Problem Definition

The high-level goal of binary code type inference is, given the executable of a compiled program, to recover a typing for the program that is as close as possible to the typing the developers used in the source code. For decompilation, it is also important that recompiling the decompiled code with the recovered typing produces binary code semantically equivalent to the original.

Binary code type inference predominantly deals with typing variables storing program data. However, in this article, we propose a unified view of binary code type inference in which an executable comprises both data and code, both of which strongly influence program behavior and can be targets of binary code type inference.

Our work shows that types are everywhere in binary code analysis. Specifically, in our unified view, many problems in binary code analysis can be seen as subproblems of binary code type inference recovering different types. For example, *disassembly* [Schwarz et al. 2002; Kruegel et al. 2004] can be seen as the subproblem of binary code type inference that types memory locations into two basic types (code and data) and may refine some code locations as the start of an instruction or a function, or belonging to a specific function. *Data type inference* [Lin et al. 2010; Lee et al. 2011; ElWazeer et al. 2013] can be seen as the combination of *variable recovery* [Balakrishnan and Reps 2007], that is, recovering the start position and size of variables, and recovering primitive types for those variables, for example, integer, float, and pointer. *Data structure identification* [Cozzie et al. 2008; Slowinska et al. 2011] recovers aggregate data types such as records and arrays. *Shape analysis* [Raman and August 2005; Jung and Clark 2009] recovers recursive types such as linked lists or trees. *Object recovery* [Fokin et al. 2011; Jin et al. 2014; Srinivasan and Reps 2014] infers classes in object-oriented programs. *Function prototype inference* [Caballero et al. 2010; ElWazeer et al. 2013] recovers function types specifying a function's parameters and return values. Even *protocol format reverse-engineering* [Caballero et al. 2007; Wondracek et al. 2008; Lin et al. 2008] can be seen as the subproblem of binary code type inference that types buffers storing protocol messages such as those passed to the `recv` and `send` functions.

Scope. To keep the scope of the article manageable, we consider works that deal exclusively with disassembly out of scope and focus on binary code type inference works that assume a (mostly) correct disassembly. Thus, works dealing with challenges such as translating machine code to assembly instructions (e.g., Schwarz et al. [2002] and Kruegel et al. [2004]), identifying function boundaries (e.g., Bao et al. [2014] and Shin et al. [2015]), and recovering jump table targets (e.g., Cifuentes and Emmerik [1999]) are out of the scope of this article. However, we do include data types related to code such as function prototypes and data embedded in code such as immediate values and offsets in the memory operands of instructions. While we do not target the recovery of indirect jump/call targets, we do include virtual table recovery since it is intrinsic to object recovery.

Typing data. In source code, a program stores data in program variables, each with an associated type. However, in binary code, program data is stored in untyped CPU registers and memory. Since architectures have few registers, most variables are stored in *memory regions*, namely, in function stack frames, heap allocations, and global sections. Thus, binary code type inference maps locations in memory regions (and registers) to high-level types such as primitives, records, arrays, class objects, and recursive types.

A variable can be characterized by the start offset in a memory region, its size, and its type. Variables in global sections can be identified by their offsets from the beginning of the section (or module). Memory locations in the stack and the heap can be reused to store different stack frames and memory allocations over time; thus, these areas need to be split into unique memory regions. A separate stack frame region is created for each program function, typically identified by the function's start address. For heap allocations, base addresses are not available statically, and may change across program executions. Therefore, a separate heap region is created for each allocation callsite, that is, program point that invokes a heap allocation function (e.g., `malloc`). Static approaches often identify these regions by the address of the allocation call [Balakrishnan and Reps 2007], but this may be problematic with custom allocators or allocation wrappers [Chen et al. 2013]. For example, a `safe_malloc` function may invoke `malloc` and check that the return pointer is not null. The callsite of `malloc` inside `safe_malloc` cannot be uniquely assigned to one type, as different `safe_malloc` callers will request allocations of different types. Dynamic approaches address this issue by using the last 3–4 callstack entries to identify heap allocations [Slowinska et al. 2010; Lin et al. 2010; Slowinska et al. 2011].

3. APPLICATIONS

Binary code type inference is a fundamental capability when the program's source code and debugging symbols are not available (e.g., malware, proprietary software) or have been lost. In this section, we explain its importance and wide impact by describing 10 applications that require or significantly benefit from binary code type inference.

(1) *Reverse engineering* – Understanding binary code without its source code or debugging symbols is a common need for analysts. Types provide rich semantics on a program's functionality and their inference is a fundamental capability for binary code reverse engineering tools such as IDA [Guilfanov 2001]. One goal of binary code type inference is to regenerate missing debug symbol tables for stripped binaries, enabling further analysis [Slowinska et al. 2011; Jacobson et al. 2011].
(2) *Decompilation* – Automatically reconstructing program source code from assembly or machine code [Cifuentes 1994; Breuer and Bowen 1994; Schwartz et al. 2013] is useful for program understanding, program optimization, and program modification. Inferring types from binary code is one of the fundamental challenges in decompilation [Mycroft 1999; Dolgova and Chernov 2009].
(3) *Binary code rewriting* – An executable can be rewritten into a different program with equivalent functionality (but with additional checks). Such rewriting is useful for porting the program to another architecture [Sites et al. 1993; Silberman and Ebcioglu 1993; Hookway and Herdeg 1997], profiling the code [Larus and Ball 1994], for optimization [Schwarz et al. 2001], and to insert an inlined security reference monitor for control-flow-integrity (CFI) [Abadi et al. 2009] and software fault isolation (SFI) [Wahbe et al. 1993; McCamant and Morrisett 2006; Erlingsson et al. 2006]. Binary code type inference helps in rewriting legacy clients (e.g., with no relocation tables) by identifying absolute memory addresses (i.e., data and code pointers) that may need to be adjusted when new code is added to the program.
(4) *Binary code reuse* – Reusing binary code enables security applications such as active botnet infiltration [Caballero et al. 2010, 2011], malware analysis [Kolbitsch et al. 2010; Zeng et al. 2013], binary code retrofitting [Zeng et al. 2013], and virtual machine introspection (VMI) [Dolan-Gavitt et al. 2011; Fu and Lin 2012]. One of the main challenges in binary code reuse is interfacing with the code to be reused, which requires recovering its prototype, including the types of the input and output variables.
(5) *Protocol reverse engineering* – The format of messages of an undocumented protocol can be recovered by analyzing the binary code implementing the protocol [Caballero et al. 2007; Wondracek et al. 2008; Lin et al. 2008; Cui et al. 2008]. Recovering the format of the protocol messages is analogous to typing the buffer holding a message received by the protocol implementation (e.g., the buffer passed to the `recv` function) and typing the buffer holding the message about to be sent on the network (e.g., the buffer passed to the `send` function). A similar problem is recovering the format of undocumented files [Lim et al. 2006].
(6) *Vulnerability detection, analysis, and prevention* – Type information is important to defend against software vulnerabilities. Inferring the location of return addresses and function pointers in the stack [Lin et al. 2010] and recovering the size of buffers (not available in binary code) [Slowinska et al. 2011] enables the

detection of buffer overflows. Using variable type information executables can be rewritten to prevent buffer overflow exploitation [Slowinska et al. 2012]. Inferring and tracking pointers throughout program execution enables the detection and debugging of use-after-free and double-free vulnerabilities [Caballero et al. 2012a].

(7) *Hooking* – A hook is a function callback point at which a particular inspection action can be performed during program execution. Hooks have been extremely valuable for malware analysis [Yin et al. 2008], VMI [Payne et al. 2008], and attack construction [Vogl et al. 2014]. Binary code type inference enables effective use of hooks when types are not available. For example, function hooks that execute code at the entry and exit points of a target function may require (e.g., when the function is internal or private) recovering the function prototype, including the types of function arguments and return value.

(8) *Memory forensics and introspection* – Extracting data of interest from live memory or a memory snapshot is a fundamental capability for VMI [Garfinkel and Rosenblum 2003] and memory forensics [Petroni et al. 2006]. Prior work shows that data type definitions can be used for building data structure signatures [Lin et al. 2011, 2012], for bridging the semantic gap in VMI [Garfinkel and Rosenblum 2003], for out-of-the-box malware analysis [Jiang et al. 2007], and for traversing OS kernel data structures to detect kernel rootkits [Carbone et al. 2009]. However, these works assume the availability of source code or debugging symbols to obtain the data type definitions. While some OSes are open source, there are proprietary OSes such as Windows that include only data type definitions in the symbols of selected modules. Furthermore, VMI and memory forensics can also be applied to application-level programs that are often proprietary [Urbina et al. 2014]. Binary code type inference (e.g., on OS kernels [Zeng and Lin 2015]) can enable these applications when source code and debugging symbols are not available.

(9) *Program data manipulation* – There are also incentives to manipulate program data in memory. One such incentive is cheating in computer games by modifying the game's memory during execution, for example, increasing unit lifetime. Prior works propose snapshot-based approaches for revealing and modifying important gaming data types such as terrain maps or unit life points [Bursztein et al. 2011; Urbina et al. 2014]. Another incentive is defeating OS defenses, for example, loading unsigned kernel drivers [Allievi 2014]. These approaches could identify those data types more accurately using binary code type inference.

(10) *Program fingerprinting* – The use of complex data structures can uniquely identify a program. It can identify malware binaries of the same family despite polymorphism [Cozzie et al. 2008] and the OS version running on a VM in a cloud environment [Gu et al. 2012, 2014]. However, these works either operate on memory snapshots [Cozzie et al. 2008] or assume the availability of the kernel data type definitions [Gu et al. 2012, 2014]. Binary code type inference can improve the accuracy over snapshot-based approaches, and enable these applications when data type definitions cannot be obtained.

This analysis not only highlights the importance of binary code type inference, but also reveals applications whose state of the art can be improved through binary code type inference. For example, program data manipulation and program fingerprinting approaches operate on snapshots and could be improved through binary code analysis. Also, most current memory forensics and introspection approaches require data structure definitions, which limit their applicability on proprietary programs.

4. APPROACHES

After discussing why to type (i.e., the motivating applications), we now discuss how to type and what to type. Tables I and II systematize 38 works for automatic binary code

Table I. Comparison of the Characteristics of Binary Code Type Inference Approaches

		Approach Characteristics													
System	Year	Venue	Binary (B) / IR (I)	Static analysis	Dynamic analysis	Combines execution results	Value-based type inference	Flow-based type inference	Instruction type sources	Function type sources	Memory access analysis	Memory graph analysis	Nested types	Merges callsites	Data in code
Mycroft [Mycroft 1999]	1999	ESOP	I	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	○
Ida [Guilfanov 2001]	2001	WCRE	B	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	○
Ew [Emmerik and Waddington 2004]	2004	WCRE	B	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	○
Rds [Raman and August 2005]	2005	MSP	B	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	○
x86sa [Christodorescu et al. 2005]	2005	PASTE	B	✓	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	○
DynCompB [Guo et al. 2006]	2006	ISSTA	B	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	○
Ffe [Lim et al. 2006]	2006	WCRE	B	✓	✗	✗	✗	✓	✗	✗	✓	✗	✓	✗	○
Divine [Balakrishnan and Reps 2007]	2007	VMCAI	B	✓	✗	✗	✗	✓	✗	✗	✓	✗	✓	✗	○
Polyglot [Caballero et al. 2007]	2007	CCS	B	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	○
Autoformat [Lin et al. 2008]	2008	NDSS	B	✗	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗	○
Wckk [Wondracek et al. 2008]	2008	NDSS	B	✓	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗	○
Laika [Cozzie et al. 2008]	2008	OSDI	B	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	○
Tupni [Cui et al. 2008]	2008	CCS	B	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	○
Dc [Dolgova and Chernov 2008]	2008	WCRE	B	✗	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	○
Dispatcher [Caballero et al. 2009]	2009	CCS	B	✗	✓	✗	✗	✓	✗	✓	✗	✗	✓	✗	○
Ddt [Jung and Clark 2009]	2009	MICRO	I	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✓	○
Bcr [Caballero et al. 2010]	2010	NDSS	B	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	○
Rewards [Lin et al. 2010]	2010	NDSS	B	✗	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	○
Ftc [Fokin et al. 2010]	2010	CSMR	B	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	○
Dde [Slowinska et al. 2010]	2010	APSYS	B	✗	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	◐
Tda [Troshina et al. 2010]	2010	SCAM	B	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	○
Tie [Lee et al. 2011]	2011	NDSS	B	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	○
Howard [Slowinska et al. 2011]	2011	NDSS	B	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	◐
SmartDec [Fokin et al. 2011]	2011	WCRE	B	✓	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	○
Recall [Dewey and Giffin 2012]	2012	NDSS	B	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	○
PointerScope [Zhang et al. 2012]	2012	NDSS	B	✗	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	◐
Artiste [Caballero et al. 2012b]	2012	TR	B	✗	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	●
Undangle [Caballero et al. 2012a]	2012	ISSTA	B	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	◐
SecondWrite [ElWazeer et al. 2013]	2013	PLDI	B	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	○
Rhk [Robbins et al. 2013]	2013	PPDP	B	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗	○
MemPick [Haller et al. 2013]	2013	WCRE	B	✗	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	○
Top [Zeng et al. 2013]	2013	CCS	B	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	○
Ym [Yan and McCamant 2014]	2014	TR	B	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	○
ObjDigger [Jin et al. 2014]	2014	PPREW	B	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	○
Lego [Srinivasan and Reps 2014]	2014	CC	B	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	○
Yb [Yoo and Barua 2014]	2014	APSEC	B	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	○
vfGuard [Prakash et al. 2015]	2015	NDSS	B	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	○
VTint [Zhang et al. 2015]	2015	NDSS	B	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	○

Note: For Boolean columns symbol ✓ means supported and symbol ✗ unsupported. For other columns, symbol ○ denotes unsupported, ◐ partial support, and ● fully supported.

type inference based on characteristics of the approach (Table I) and the types that each approach infers (Table II).

To select these 38 works, we first made a list of seed papers that we knew related to binary code type inference. To identify more works, we then examined the papers published since 1998 in the venues in which those seed papers were published, and performed searches on engines such as Google Scholar for binary code type inference terms. For every work on binary code type inference or its applications, we examined its references for missing papers. We selected works that present techniques and approaches for automatic binary code type inference as well as works whose goal may be a different application (e.g., protocol reverse-engineering, vulnerability and exploit analysis), but that frame their approach as binary code type inference or propose techniques later adapted to binary code type inference. We do not include all works in those other areas because we systematize only binary code type inference. We limit the survey to papers published in peer-reviewed venues and technical reports from academic institutions. We do not include tools, but rather the works describing their techniques (e.g., IDA [Guilfanov 2001] and CODESURFER [Balakrishnan and Reps 2007]).

Papers are identified by their system name if available, otherwise by the author's initials. The approaches are sorted by publication date, creating a timeline of the development of the field. These 38 works have been published in 18 venues and 2 technical reports (TR). The top 3 venues for publishing papers on binary code type inference are NDSS (10 papers), WCRE (6), and CCS (4). The field of binary code type inference is multidisciplinary, with papers appearing in venues from different areas such as security (e.g., NDSS, CCS), reverse engineering (e.g., WCRE), systems (e.g., OSDI), programming languages (e.g., PLDI, CC), and software engineering (e.g., ISSTA, PASTE). Works focusing on reverse-engineering and decompilation are more likely to be published at specialized venues such as WCRE. Works published at top security conferences typically target other security applications. One goal of Tables I and II is helping readers quickly locate works dealing with subsets of binary code type inference regardless of the venue in which they appeared.

We split the discussion of both tables into two sections. The rest of this section details the approach characteristics (Table I) with each subsection describing a subset of table columns. Then, in Section 5, we detail the types inferred by each approach (Table II).

4.1. Input

Column: *Binary (B) / IR (I)*

Strictly speaking, the input to binary code type inference should be just the binary code. The vast majority of approaches (36/38) fall into this case. However, we include two exceptions that take as input an intermediate representation (IR). MYCROFT is the first approach for binary code type inference. It takes as input an RTL IR obtained from BCPL, an untyped predecessor of C. A front-end from x86 to RTL is available in the Boomerang decompiler [Boomerang 2004]. DDT takes as input LLVM IR obtained from the program's source code. However, the authors argue that they do not use any type information in the IR and present the techniques as if operating on binary code. It has been cited by follow-up work and constitutes the only example on abstract type recovery (Section 5.8). An x86 to LLVM front-end such as the one in SECONDWRITE would, in theory, enable DDT to work on binary code.

In Guo et al. [2006], two tools are presented: one operating on x86 code (DYNCOMPB) and the other on Java bytecode (DYNCOMPJ). While the type synonym inference problem (Section 5.6) addressed in this work makes sense even with the additional type information in bytecode, our focus is on binary code type inference. Since both tools use the same approach, we include DYNCOMPB, but exclude DYNCOMPJ.

4.2. Type of Analysis

Column: *Static analysis; Dynamic analysis; Combines execution results*

Binary code type inference approaches can be classified by the type of analysis performed: static analysis, dynamic analysis, and combinations of both. Static analysis examines the disassembled binary code, while dynamic analysis examines program executions by running the program on given inputs.

Table I shows that 21 out of 38 works use static analysis, 21 use dynamic analysis, and 4 combine static and dynamic analysis (WCKK, DDT, BCR, TIE). The advantages of dynamic analysis are accuracy and simplicity. Both advantages come from the fact that memory addresses, operand values, and control-flow targets are known during execution. Also, static analysis requires proper disassembly of the code, which may be difficult in some situations, for example, obfuscated code. On the other hand, static analysis has better code coverage. While dynamic analysis examines one execution at a time, static analysis can analyze all program paths without the challenge of obtaining an input suite that executes all program paths, and without re-executing the program on all those inputs, which can be slow.

Of the dynamic approaches, only 3 combine type inference results from multiple executions (HOWARD, BCR, ARTISTE). Combining multiple executions can improve dynamic binary code type inference results because variables may only be used (and thus their types inferred) in some program paths. Also, different types may be inferred for the same variable in different executions and combining results from multiple executions enables assigning a more refined inferred type for each variable.

One reason for the use of dynamic approaches, despite their limited coverage, may be that many applications do not require typing the whole program, but only certain data structures or functions of interest. For example, game hacking recovers only map and unit data structures [Bursztein et al. 2011; Urbina et al. 2014] and binary code reuse may need only the prototype of the functions to be reused [Caballero et al. 2010].

4.3. Value-Based Type Inference

Column: *Value-based type inference*

Approaches can infer types by examining the values stored in registers and memory (*value-based*), by propagating types from available type sources (*flow-based*), or by combining both. Value-based type inference does not require examining the code, only the content of memory/registers. However, it is based on heuristics that can introduce errors. It is typically used only for identifying pointers and strings [Raman and August 2005; Cozzie et al. 2008; Haller et al. 2013; Srinivasan and Reps 2014]. For example, the `strings` Unix command identifies strings as sequences of bytes of some minimum length that could represent printable characters in popular encodings (e.g., ASCII, UTF-16). For pointers, these approaches check if 4 consecutive bytes (8B in 64b architectures) in live memory (or a register) form a value corresponding to the address of a live memory byte, that is, an address in global data sections, live heap allocations, or live stack frames. If so, those bytes could store a pointer variable. Unfortunately, those bytes could also correspond to an integer variable, (part of) a string, or be unrelated. Fine-grained information about what constitutes live memory increases the accuracy, for example, tracking heap allocations and the current stack height is more accurate than considering alive all bytes in pages mapped to the program.

In total, 8 out of 38 use value-based inference, which is also commonly used together with memory graph analysis (Section 4.6).

4.4. Flow-Based Type Inference

Column: *Flow-based type inference; instruction type sources; function type sources*

Flow-based type inference is used by 32 out of 38 approaches. At a high level, variables are assigned types based on how the program uses them. The program code introduces two kinds of constraints: type propagation and type inference. Type inference constraints assign types to variables based on how they are used. There exist two major sources of type inference constraints: instructions that operate on their data, for example, arithmetic instructions, and calls to external functions for which their prototype is known [Ramalingam et al. 1999; Guilfanov 2001]. For example, `imul %esi,%edi` multiplies the values in 32b registers ESI and EDI, which constrains the variables in ESI and EDI as numbers. Similarly, a call to the standard C library function `size_t strlen(const char *)` constrains the location storing the parameter at function entry as a string pointer, and the return value at function exit as an unsigned integer.

Type propagation constraints are produced by instructions that simply move data (e.g., `mov`, `push`, `pop`) or assignment statements in the IR. These instructions introduce constraints of the type that the destination should have the same type as the source. However, it is important to note that different works handle these instructions differently. Some works assume that such data movement instructions introduce some type information regarding size. For example, given an instruction `mov %ecx,(%edi)`, which moves the content of register ECX into the memory location pointed by register EDI, some approaches (e.g., TIE) may add the constraints that, after the instruction executes, the content of the memory location will have the same type as the content of ECX, which will be a 32b type since that is the size of ECX and the destination memory range. However, a program may move a variable from one location to another without regard to its size. For example, a program could move a 32b integer variable one byte at a time using four instructions, or in a 32b architecture it could move a C `long long` variable (64b) using two instructions that move 4B at a time. Furthermore, there exist functions (e.g., `memcpy`) that copy a source memory range to a destination memory range without regard to the internal structure of the source memory range. Thus, the instructions that implement `memcpy` may move chunks without respecting the internal field structure, for example, one loop iteration may move 4B that correspond to the last 2B of one variable and two `char` variables stored next to it. This does not matter, as the semantics of the function guarantee that the destination range contains the same data as the source range after the function returns. Solutions to this issue include not adding a size constraint to instructions that move data, ignoring propagation inside memory copy functions, and establishing type constraints at a byte-level granularity.

Note that an instruction may introduce both typing and propagation constraints. In the earlier example, `mov %ecx,(%edi)`, in addition to the type propagation constraints, some approaches add a type inference constraint that register EDI must hold a pointer since its value is being dereferenced to access memory.

Flow-based approaches introduce type propagation constraints as they analyze code paths, either statically or dynamically, until the untyped variables flow into an instruction operand or function parameter for which the type is known. Similarly, when the output operand of an instruction or the return value of a function has a known type, it is propagated forward, typing the variables that the type flows into.

Static memory analysis. Static flow-based approaches need a points-to (or alias) analysis to identify if a memory location being overwritten is later read, so that type propagation constraints can be updated through memory accesses. Dynamic approaches do not require such analysis, as memory addresses have concrete values at runtime. Early static approaches such as MYCROFT and IDA do not support alias analysis.

x86SA uses an alias analysis for strings. FFE and DIVINE leverage value-set-analysis (VSA) [Balakrishnan and Reps 2004], a sound flow-sensitive, context-sensitive, interprocedural, abstract-interpretation algorithm that overapproximates the set of numeric values and addresses that each register and memory location holds at a program point. VSA enables the tracking of the flow of values through indirect memory accesses. TIE uses a variant of VSA called DVSA. SECONDWRITE introduces the concept of best-effort pointer analysis, which sacrifices soundness for improved performance (further discussed in Section 8).

Online and offline solving. Flow-based dynamic approaches (e.g., REWARDS, POINTERSCOPE, UNDANGLE) implement constraint propagation using tainting, where constraints are solved online, as soon as a type source or a type sink is reached. Thus, at any point during the execution, it is possible to output the currently inferred types for a region, for example, when heap allocations are freed or when a function returns. Static approaches such as MYCROFT, TIE, and RHK use a different approach that outputs constraints as code paths are analyzed and solves them offline using a custom solver, that is, at the end of the analysis when all constraints are available. It is possible for both static and dynamic approaches to output a solution for which no bytes can be typed, for example, all bytes are assigned bottom from the primitive type lattice (Section 5.1). Offline approaches often mention this situation (MYCROFT, TIE, RHK) and some (MYCROFT, TIE) describe additional mechanisms to identify the constraints causing the failure and handling them in a special way, for example, removing them or converting them to unions and casts. Online approaches do not typically mention this situation, which could indicate that it does not affect them as frequently as offline approaches. This could be due to online approaches limiting conflict propagation by performing more localized constraint propagation (e.g., one path at a time, outputting types for a heap region when it is deallocated).

Unification. In unification, instructions that move data (*dst* $\leftarrow$ *src*) unify the types of *dst* and *src* so that not only the type of the *src* affects the *dst*, but also the other way around. This is problematic when *dst* is a location used for temporary storage that may hold variables of different types over time. Static approaches handle this issue by employing single static assignment (SSA) notation, so that writing to a location creates a fresh variable. Dynamic approaches that do not use an IR handle this issue in different manners. REWARDS disables unification when *dst* is a register, but not in other problematic situations such as stack locations, in which a type lifetime is associated based on the data lifetime. POINTERSCOPE unifies only on source operands, overwriting the *dst* type with the *src* type after an instruction executes. Independent evaluation has compared these two dynamic approaches [Caballero et al. 2012b], showing how unifying only on source operands improves typing accuracy and performance.

Instruction type sources. Twenty approaches derive types using instruction type sources. We have identified two issues that may point to the difficulty of extracting type information at the instruction level. First, some instruction type inference constraints seem too general. For example, MYCROFT considers the `xor` operands to be integers, but encryption algorithms may XOR any data type. This issue also affects other bitwise operations, for example, `and`, `or`. In addition, differences on the type inference constraints for the same instruction are not uncommon, leading to differences in the typing results even if the same algorithm is used. For example, TIE authors consider the operand of unary negation a signed integer, but YM authors consider that operation on unsigned values perfectly legal. Similarly, SECONDWRITE authors consider operands in a left/right shift of the same type, and POINTERSCOPE authors consider them to be integers, but they may not even be of the same size. We believe that there is a need for more thorough evaluation of type inference rules for instructions, for example, quantifying the errors and type conflicts that different constraints for the same instruction introduce.

Function type sources. External function calls provide a wealth of *semantic types* such as IP addresses and timestamps. For example, REWARDS defines 150 types from 84 standard library functions. Among the 38 approaches, 10 use function sources to derive semantic types. One problem in those approaches is that semantic types are not included in the type lattice. Thus, it is unclear how to relate them to other types, for example, whether they are synonyms of other types. Note also that mapping parameters and return values to registers and stack locations requires knowing the calling convention used by the compiler (e.g., obtained from the library symbols), assuming a specific convention (e.g., `cdecl` in REWARDS), or inferring the function prototype (Section 5.9).

4.5. Memory Access Analysis

Column: *Memory access analysis*

Memory access patterns reveal the variable layout of aggregate types such as records and arrays [Mycroft 1999; Ramalingam et al. 1999]. This intuition has been widely applied to binary code to recover variables by examining the offset of memory accesses with respect to a base pointer and the size of the access. A base pointer can belong to the heap, the stack, or a global region. The base pointer for heap regions is the pointer returned by the allocation function (e.g., `malloc`), and for stack frames the value of the stack register (e.g., `ESP`) at the function's entry point. For global variables, base pointers can be identified using relocation tables and fields in the executable's header [Slowinska et al. 2010]. Intuitively, all memory accesses in a memory region should (directly or indirectly) derive from its base pointer. For example, the sequence `call <malloc>; movl $0x3, 0x4(%eax)` indicates the existence of a 32b variable at offset 4 in the heap allocation whose address was returned by `malloc` in register EAX. The size of the variable comes from the length of the `movl` memory operand.

Limitations. Although memory access pattern–based variable recovery is very popular, it also has some known limitations. An obvious limitation is that it can only recover fields that the program accesses (or accessed in an execution for dynamic approaches). There are three other less-obvious limitations. One limitation is that a memory region can be accessed without considering its layout. For example, the function `memcpy` copies a memory region in word-size chunks, regardless of the structure of that region. To address this problem, DDE and HOWARD propose giving preference to nonregular accesses and strides not equal to the word size.

Another limitation is that it can only identify variables of at most the word size. For example, storing a 64b (double) float variable in memory requires two instructions in a 32b architecture, which would incorrectly recover two consecutive 4B variables, as discussed in Section 4.4.

Yet another limitation is that variables in an aggregate type may be accessed using different base pointers. For example, variables in a record local variable may be accessed through the base pointer of the stack frame, or through the base pointer of the record structure. If accessed only through the base pointer of the stack frame, the member variables of the record will be correctly identified, but the record that stores them becomes indistinguishable. A similar issue may occur with nested aggregate types (e.g., a record inside another record; see Section 4.7).

Column *memory access analysis* in Table I marks whether an approach analyzes memory accesses to identify the location and size of variables. Out of 38 works, 15 use memory access analysis. Among them, protocol reverse engineering works heavily use this approach for identifying message fields.

4.6. Memory Graph Analysis

Column: *Memory graph analysis*

The analysis of the points-to relationships between memory regions can reveal recursive types such as lists and trees (Section 5.5). To analyze those relationships, dynamic approaches may build *memory graphs*, directed graphs in which nodes correspond to memory regions (e.g., heap allocations, loaded modules) and edges are pointers between those regions. Note that if a type (e.g., a class) is allocated multiple times, each instance (object) has a node in the memory graph. Multiple memory graphs can be built at different execution times as heap regions are allocated/deallocated and modules loaded/unloaded.

There are 6 approaches, all dynamic, that build memory graphs (RDS, LAIKA, DDT, REWARDS, ARTISTE, MEMPICK). Static approaches do not build memory graphs since concrete region start addresses are not available statically, which makes it difficult to map allocations to deallocations. Furthermore, if regions are created in loops, it may not be possible to statically infer how many nodes the allocation creates, that is, if the number of loop iterations depends on external input.

All 6 approaches identify the start address and size of nodes by tracking heap allocations/frees throughout program execution, except LAIKA, which applies a (less accurate) machine-learning approach that infers this information directly from the memory snapshot. Recent work [Urbina et al. 2014] adds loaded modules to the memory graph and shows that it is possible to extract live heap allocations and loaded modules from a memory snapshot by introspecting the OS memory manager data structures without requiring the approximations used by LAIKA.

For edges, both value-based (RDS, LAIKA, DDT, MEMPICK) and flow-based (REWARDS, ARTISTE) pointer type inference are used. Nodes are annotated with their inferred type, which can be the allocation callsite (RDS, REWARDS), the merged callsites in Section 4.8 (DDT, ARTISTE), or a type assigned to allocations processed by the same pointer manipulation instructions (MEMPICK).

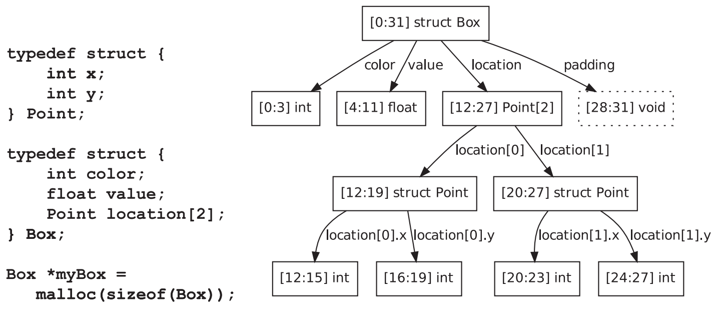
The memory graph can be output at selected times (LAIKA, REWARDS), quiescent points without activity (MEMPICK), before and after each function (DDT), periodically (ARTISTE), or its information can be accumulated over time (RDS).

4.7. Nested Types

Column: *Nested types*

A byte in a memory region can belong to multiple nested types, for example, to a primitive integer type and to several aggregates such as an integer array, and a record storing the integer array. There are 8 works that capture nested types using diverse representations. The most common representation are trees used by 5 works (DIVINE, WCKK, AUTOFORMAT, DISPATCHER, ARTISTE). Other representations include FFE, which captures nested types in hierarchical finite state machines (HFSMs), and HOWARD, which provides a textual representation of nested types, for example, an array inside a record.

In the tree representations, the root node typically corresponds to a memory region (stack frame, heap allocation, loaded module) and the other nodes to variables with a specific type. The leaf nodes correspond to primitive types and the internal nodes to aggregate types such as records, arrays, or class objects. An example tree representation is illustrated in Figure 2. On the left, it shows the data structure definition of a `Box`



```
typedef struct {
    int x;
    int y;
} Point;

typedef struct {
    int color;
    float value;
    Point location[2];
} Box;

Box *myBox =
   malloc(sizeof(Box));
```

Fig. 2. Data structure definition and example tree that captures its variable layout.

type comprising 3 variables: an integer, a float, and an array of two `Point` types. On the right, it shows the tree for the heap allocation pointed to by the `myBox` pointer. Depth one captures the variables in the `Box` type; the rest captures the array structure. Each edge is annotated with a variable name (not available in the binary code) and each node with the range of bytes that it occupies, and its type. To account for the compiler aligning data structures to 64b, a fake `padding` variable has been added.

Divine first used trees and called them ASI trees, but only defined the variable size and the relative ordering between variables. Protocol reverse-engineering works [Wondracek et al. 2008; Lin et al. 2008] extended the trees by adding the variable/field type and the start offset (to account for variables not accessed by the program).

4.8. Merges Callsites

Column: *Merges callsites*

The same type can be allocated at different program points. Initially, each heap region is considered a separate recovered type identified by the callsite. Merging callsites of the same type can improve accuracy by combining data from the different callsites, for example, adding variables only recovered for one callsite, refining types for variables at the same offset, and performing the union of recovered methods for class callsites.

Only three works identify and merge callsites of the same type. Ddt identifies callsites of the same type if allocations from those callsites are accessed by the same functions. ObjDigger merges two recovered classes if they use the same constructor or the same virtual table. Artiste merges format trees from different callsites in two situations. First, it merges callsite trees that are pointed to by the same pointer at different points in time. Second, it clusters callsite trees based on their format and profiling information on the instructions that operate on them. We have not found any work that attempts to identify that two aggregate types in different stack frames, or in a combination of heap allocations and stack frames, are of the same type.

4.9. Data in Code

Column: *Data in code*

As explained earlier, we consider disassembly out of the scope of this article, which includes separating code from data and typing code locations as instructions and functions. Instead, this section deals with special cases in which code locations may be typed as containing data. In particular, memory locations inside an instruction can be further subtyped as data if they contain immediate values or an offset value in a memory operand, since those values may be of different types such as integer or pointer. For example, instruction `mov %eax,0x7c90ec15(%esi)` moves the content of register EAX to an entry in a global table starting at address 0x7c90ec15. This instruction occupies 7B in memory, of which the last 4B contain the offset 0x7c90ec15. All 7B bytes are typed by the disassembler to be an instruction, but the last 4B could be subtyped as a pointer. In contrast, in instruction `mov %eax,0xc(%esi)`, the offset value 0xc could be subtyped as a short integer. Immediates can be handled similarly. Instruction `mov $0x77c11fe8,%edi` occupies 5B, of which the last 4B correspond to the immediate 0x77c11fe8, which could be typed as a pointer, but in instruction `mov $0x1,0x8(%esi)`, the immediate would be a short integer.

A popular approach to identify pointers to global data is using relocation tables (e.g., Zhang et al. [2013]), which contain the locations inside instructions that need to be patched if the module is relocated. In Table I, we assign partial data in code support to works that identify code pointers using relocation tables, and full support to those that also identify nonpointer types in immediates and offsets.

5. INFERRED TYPES

This section discusses Table II, which captures the types inferred by each approach. In the table, ● indicates complete support, ◐ partial support, and ○ lack of support. A quick glance shows that there exist many different types that may be targeted by binary code type inference and that support is very sparse. This is one important conclusion of this work. No single work tries to recover all types; some types have little support (e.g., type synonyms, unions); and some, such as abstract types, received significant less attention, constituting good candidates for future work. One reason for the sparse support is that some works recover only the types needed by their applications, for example, pointers for vulnerability and exploit analysis [Zhang et al. 2012; Caballero et al. 2012a] and function prototypes for binary code reuse [Caballero et al. 2010]. However, even works focusing on reverse engineering and decompilation show sparse support. Regarding the final output of the type inference, most approaches produce a single type for the identified variable, and one (TIE) outputs a type range.

5.1. Primitive Types

Column: *Primitive type lattice; Integer; Floating point; Pointer & Pointer target*

Primitive types are the smallest type units and are provided as built-in types by the programming language, for example, integer, pointer, char, bool, float, double. A primitive type lattice captures the refinement relationships between primitive types, which dictates how an approach unifies variables of different types. Unfortunately, only 5 works that recover primitive types detail their primitive type lattice. The rest simply list the inferred primitive types, without their refinement relationships.

Figure 3 shows the primitive type lattices of TIE and ARTISTE, the most complete that we found. In the lattices, $\top$ represents an unknown type and all types are implicitly connected to $\perp$, which represents a type conflict. The goal is to infer refined primitive types, that is, as far down the lattice as possible without reaching $\perp$.

Both lattices are largely similar, but contain differences due to additional types supported by ARTISTE, for example, floating-point and 64b data, and some author decisions, for example, separating code from data and considering pointers to be number subtypes (i.e., `ptr32` refines `num32`). The latter change makes arithmetic operations (e.g., `add`, `sub`) to operate on numbers, regardless of whether performing integer or pointer arithmetic [Caballero et al. 2012b]. This is a simplification over previous approaches that use disjunctive type constraints for arithmetic operations (since the operands could be integers or pointers) [Mycroft 1999; Lee et al. 2011]. It also helps with zero constants, which could be both integer or NULL pointer, and can be assigned `num32` in the meantime.

Table II. Comparison of the Inferred Types of Binary Code Type Inference Approaches

	Primitive Types				**Classes**			**Other Types**							
System	**Primitive type lattice**	**Integer (signed & unsigned)**	**Floating point**	**Pointer & Pointer target**	**Class methods**	**Class hierarchy**	**Virtual tables**	**Records**	**Arrays**	**Strings**	**Recursive types**	**Type synonyms**	**Unions**	**Abstract types**	**Function types**
Mycroft [Mycroft 1999]	✗	●	○	●	○	○	○	●	◐	○	○	○	◐	○	◐
Ida [Guilfanov 2001]	✗	◐	◐	●	○	○	○	◐	◐	◐	○	○	◐	○	◐
Ew [Emmerik and Waddington 2004]	✗	○	○	○	○	◐	○	○	○	○	○	○	○	○	○
Rds [Raman and August 2005]	✗	○	○	◐	○	○	○	◐	○	○	◐	○	○	○	○
x86sa [Christodorescu et al. 2005]	✗	○	○	○	○	○	○	○	○	●	○	○	○	○	○
DynCompB [Guo et al. 2006]	✗	○	○	○	○	○	○	○	○	○	○	●	○	○	○
Ffe [Lim et al. 2006]	✗	○	○	○	○	○	○	◐	●	◐	○	○	○	○	○
Divine [Balakrishnan and Reps 2007]	✗	○	○	○	○	○	○	◐	●	◐	○	○	○	○	○
Polyglot [Caballero et al. 2007]	✗	○	○	○	○	○	○	○	◐	◐	○	○	○	○	○
Autoformat [Lin et al. 2008]	✗	○	○	○	○	○	○	◐	◐	◐	○	○	○	○	○
Wckk [Wondracek et al. 2008]	✗	○	○	◐	○	○	○	○	◐	◐	○	○	○	○	○
Laika [Cozzie et al. 2008]	✗	○	○	◐	○	○	○	◐	○	●	◐	○	○	○	○
Tupni [Cui et al. 2008]	✗	○	○	○	○	○	○	◐	◐	◐	○	○	○	○	○
Dc [Dolgova and Chernov 2008]	✗	●	●	◐	○	○	○	◐	○	○	○	○	○	○	○
Dispatcher [Caballero et al. 2009]	✗	○	○	○	○	○	○	○	◐	◐	○	○	○	○	○
Ddt [Jung and Clark 2009]	✗	○	○	◐	○	○	○	○	○	○	●	○	○	◐	○
Bcr [Caballero et al. 2010]	✗	◐	○	◐	○	○	○	○	○	○	○	○	○	○	●
Rewards [Lin et al. 2010]	✗	◐	●	◐	○	○	○	●	○	●	◐	○	○	○	○
Ftc [Fokin et al. 2010]	✗	○	○	○	○	◐	●	○	○	○	○	○	○	○	○
Dde [Slowinska et al. 2010]	✗	○	○	◐	○	○	○	●	●	◐	○	○	○	○	○
Tda [Troshina et al. 2010]	✗	●	●	●	○	○	○	●	●	◐	○	○	○	○	○
Tie [Lee et al. 2011]	✓	●	○	●	○	○	○	◐	○	○	○	○	◐	○	◐
Howard [Slowinska et al. 2011]	✗	○	○	◐	○	○	○	●	●	◐	○	○	○	○	○
SmartDec [Fokin et al. 2011]	✗	●	●	●	●	◐	●	●	●	◐	○	○	○	○	○
Recall [Dewey and Giffin 2012]	✗	○	○	○	◐	○	●	○	○	○	○	○	○	○	○
PointerScope [Zhang et al. 2012]	✓	◐	○	◐	○	○	○	○	○	○	○	○	○	○	○
Artiste [Caballero et al. 2012b]	✓	●	●	●	○	○	○	●	◐	◐	●	○	◐	○	○
Undangle [Caballero et al. 2012a]	✓	○	○	◐	○	○	○	○	○	○	○	○	○	○	○
SecondWrite [ElWazeer et al. 2013]	✗	◐	◐	●	○	○	○	●	●	◐	◐	○	○	○	●
Rhk [Robbins et al. 2013]	✗	●	○	●	○	○	○	●	○	○	◐	○	○	○	○
MemPick [Haller et al. 2013]	✗	○	○	◐	○	○	○	◐	○	○	●	○	○	○	○
Top [Zeng et al. 2013]	✗	○	○	●	○	○	○	○	○	○	○	○	○	○	◐
Ym [Yan and McCamant 2014]	✗	◐	○	○	○	○	○	○	○	○	○	○	○	○	○
ObjDigger [Jin et al. 2014]	✗	○	○	◐	●	◐	●	◐	○	○	○	○	○	○	○
Lego [Srinivasan and Reps 2014]	✗	○	○	○	●	●	○	○	○	○	○	○	○	○	○
Yb [Yoo and Barua 2014]	✗	◐	◐	●	◐	◐	●	●	●	○	○	○	○	○	●
vfGuard [Prakash et al. 2015]	✗	○	○	◐	◐	○	●	○	○	○	○	○	○	○	○
VTint [Zhang et al. 2015]	✗	○	○	◐	◐	○	●	○	○	○	○	○	○	○	○

Note: Symbol ○ denotes a type is not supported (not inferred), ◐ denotes partial support, and ● denotes full support.

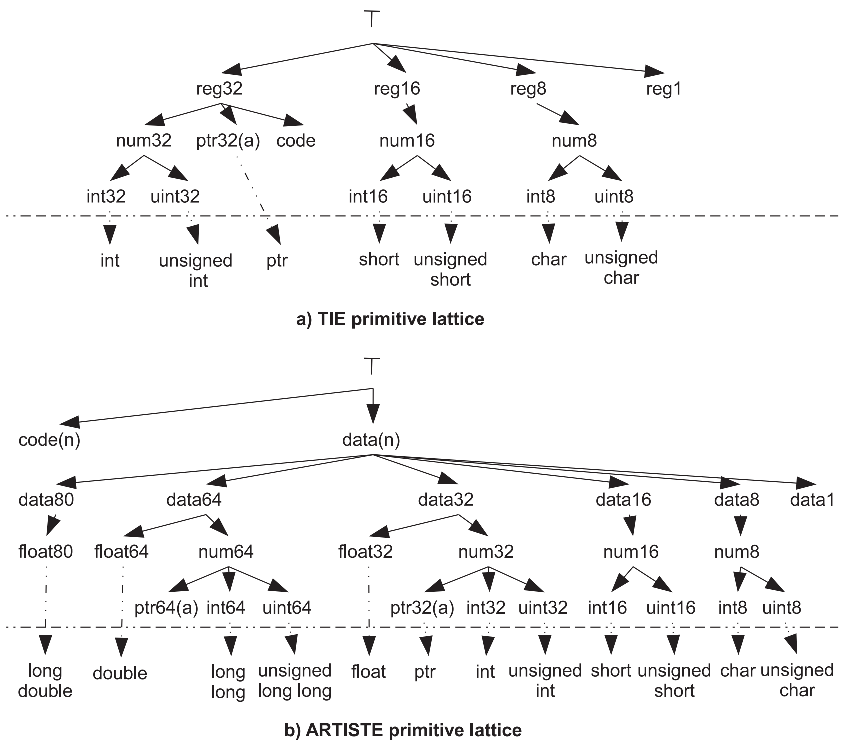


Fig. 3. Primitive type lattices in TIE and ARTISTE. Above the horizontal line are the inferred primitive types and below the mapping to 32b C types. All leaf types above the horizontal line are implicitly connected to ⊥. We have slightly modified the TIE type names to facilitate the comparison between both lattices. The original TIE lattice appears in Lee et al. [2011].

The dotted line marks the translation used from the inferred primitive types to C types in a 32b architecture. Note that some inferred types could correspond to different source types with identical representation. For example, compilers often represent `bool` using 1B,[1] thus they can only be distinguished from `char` by checking that their value is always zero or one.

The approaches studied in Table II recover three primitive types: integers, floats, and pointers. Other primitive types such as `char` and `bool` are considered 1B integers. Pointers are the most commonly inferred type since they are a prerequisite to infer aggregate types such as records, arrays, and recursive types. They are also the target of works on vulnerability and exploit analysis [Zhang et al. 2012; Caballero et al. 2012a]. In Table II, we assign full pointer support to approaches that differentiate pointers by their target type, that is, they distinguish between a `struct a*` and a `struct b*`, and partial support otherwise. Works that fully support pointers parameterize them with a single target type, except ARTISTE, which uses a set. A set is useful for pointers that point to instances of the same type allocated at different callsites, and for class inheritance, in which a pointer may point to objects of the parent and child classes (Section 5.2).

In Table II, an approach gets full integer support if it infers integers of different sizes (1B, 2B, 4B, 8B) and signedness, and partial if it recovers only one of those properties. YM authors argue that inferring signedness is highly challenging due to the absence

[1]The C++ specification does not mandate the `bool` representation.

of casts in binary code and that few instructions truly manifest signedness [Yan and McCamant 2014]. Thus, many variables are inferred as generic numbers (e.g., num32) rather than their more refined signed and unsigned subtypes.

The x86 floating point registers can operate on single precision (32B), double precision (64B), and extended precision (80B) floats. We assign full float support to approaches that infer floats of different sizes and partial if they do not differentiate sizes. Of the static approaches, only SecondWrite performs height analysis of the floating point stack.

5.2. Classes

Column: *Class methods; Class hierarchy; Virtual tables*

Classes in C++ object-oriented programs are also a popular target of binary code type inference. Existing approaches focus on four main challenges to class recovery: first, recovering the class methods, for example, constructors and destructors; second, recovering the class hierarchy lattice produced by inheritance, which manifests in the embedding of an object of each superclass inside the object of the subclass; third, recovering virtual methods and the virtual tables used to dispatch them; and fourth, recovering the data member layout. This section focuses on the first three challenges. The last one is identical to the recovery of record layouts discussed in Section 5.3. SmartDec and Yb also tackle the recovery of exception-handling mechanisms, but we consider this part of structural analysis and out of the scope of this article.

Important properties of C++ classes, such as where to pass the `this` pointer, how to dispatch virtual methods, and how to combine parent classes, are dictated by the ABI. Most Unix compilers, including modern versions of g++, follow the Itanium ABI, while most Windows compilers follow the MSVC ABI. Seven approaches (Ew, Ftc, SmartDec, Recall, ObjDigger, VTint, vfGuard) assume MSVC ABI and two (Lego, Yb) Itanium ABI. However, most proposed techniques could be applied to both ABIs with some effort.

Ew first proposed recovering class hierarchy information leveraging runtime type information (RTTI) and Windows message maps. An RTTI data structure is emitted by the compiler for each polymorphic class (i.e., with a virtual method) to support the `typeid` and `dynamic_cast` C++ operators. It includes the class name, inheritance hierarchy, and parts of the class layout. However, it is not available for classes without virtual methods, it is often absent in commercial software, and it does not provide information on nonvirtual methods. More recent approaches do not assume RTTI availability (the exception being Yb), but leverage it if present.

In both MSVC and Itanium ABIs, a pointer to the RTTI data structure precedes the virtual table. Thus, identifying virtual tables reveals RTTI structures as well. Approaches that recover virtual tables essentially scan the data sections for arrays of function pointers using value-based type inference or leverage the executable's relocation and export tables.

Ftc proposes a technique (also used by SmartDec) to recover a polymorphic class hierarchy by tracking virtual table pointer modifications in constructors and destructors. The technique assumes that each polymorphic class inherits from at most one parent class and does not work for nonpolymorphic classes. SmartDec proposes to identify nonpolymorphic methods and assign them to a specific class by monitoring the value of the `this` pointer passed in ECX to the method (assuming MSVC's `thiscall` calling convention). Tracking the `this` pointer is also used by Recall with polymorphic classes and by ObjDigger (statically) and Lego (dynamically) for all methods. Lego proposes a class hierarchy recovery technique based on destructor sequences.

In Table II, we assign full class method support to an approach if it identifies polymorphic and nonpolymorphic class methods and it does not require RTTI, and partial support if it requires RTTI or only identifies polymorphic methods. Similarly, we assign full class hierarchy support if the hierarchy of both polymorphic and nonpolymorphic classes is recovered without RTTI, and partial support if RTTI is required or it only recovers polymorphic class hierarchies. We assign full virtual table support to all approaches recovering virtual tables. We also give partial pointer support to approaches that track the propagation of `this` pointers.

Remaining challenges. There exist a number of challenges not fully solved by current approaches. These include identifying static methods (which do not use the `this` pointer), inlined constructors and destructors, elimination of virtual table references in constructors through optimization, distinguishing composition from inheritance, recovering public/private/protected method attributes, and handling templates.

5.3. Records and Arrays

Column: *Records; Arrays*

Record types are derived by combining multiple primitive types, for example, a record comprising two integers created using the C `struct` keyword. They are often informally called *data structures*. Arrays are a common built-in type for sequences of elements of the same type. In records, each element (or field) may have a different type that needs to be inferred. In arrays, all elements have the same type; once an element's type is inferred, the type of the array is known.

One commonality between records and arrays is that, to access one of their elements, a program often uses a base pointer and an offset. This is particularly true when they are allocated in the heap. Therefore, memory access analysis (Section 4.5) has been proposed to infer records and arrays both statically [Ramalingam et al. 1999] and dynamically [Slowinska et al. 2010]. To differentiate arrays from records, those approaches leverage that each element of an array has the same type (and size). Thus, regularly spaced offsets accessed using the same base pointer, (e.g., {0,4,8,12}) may indicate an array with a *stride* of 4B, that is, with 4B elements. To differentiate a record with 4 integer variables from an array of 4 integers, they leverage that arrays are more commonly processed in loops compared to records.

One challenge common to records and arrays is estimating their size if they do not have an explicit size field, for example, in C programs. An incorrect estimate may miss the last elements of the record/array or may include unrelated data stored next to them. Three approaches used to estimate the size are (1) approximate it by the largest offset that the base pointer is used with plus the size of that access; (2) bound the size by the starting address of the next variable inferred in the memory that follows; and (3) for heap-allocated records/arrays, bound their size by the size of the heap allocation that contains them.

Column *Records* assigns partial support to approaches that recover the variable layout of records and full support to approaches that also type variables with primitive types. Column *Arrays* assigns partial support to approaches that analyze strides in memory accesses to identify arrays and the size of their elements, and full support if they recover nested arrays.

Some open challenges remain. One is accurately identifying records and arrays in global memory regions and the stack. This is challenging with global memory regions because a program can directly access their elements with a fixed virtual address instead of using a base pointer and an offset. In the stack, their elements can be accessed through their base pointer, but also through the base pointer of the stack frame, which may differ if there is another local variable in the stack before the beginning of the record/array. Other open challenges include identifying multidimensional arrays, arrays accessed using SIMD instructions, and padding fields introduced by the compiler for alignment.

5.4. Strings

Column: *Strings*

Strings are sequences of characters with some particular encoding such as ASCII or Unicode. In C, strings are represented as null-terminated arrays of characters. Many other string representations exist. For example, strings can be implemented as a class (e.g., `basic_string` C++ class) or with abstract data types that define the operations, but hide the representation (Section 5.8). Strings are difficult to differentiate from their underlying representation, for example, from arrays of characters. In Table II, we assign partial string support to approaches that identify arrays. We assign full support to approaches that distinguish between arrays and C strings using value-based type inference (LAIKA), function type sources (X86SA, REWARDS), or instruction type sources (REWARDS). None of the approaches identifies other string representations.

Dynamic arrays. Dynamic arrays are variable-length buffers that may change size during execution through reallocation. They do not exist as such in C/C++ source code but are commonly used to implement other types such as strings or the C++ vector class. For example, a common string representation is a record holding the maximum string size, the current string size, and a pointer to a dynamic array storing the characters. Only ARTISTE identifies dynamic arrays.

5.5. Recursive Types

Column: *Recursive types*

Recursive types are aggregate types storing recursive pointers (pointers to the same aggregate type storing them) forming shapes such as lists and trees. Note that a global or local pointer to another pointer (e.g., `char** a`) is a primitive type rather than a recursive type since it does not belong to an aggregate type. In Table II, we assign partial support to approaches identifying recursive pointers (RDS, LAIKA, REWARDS, RHK, SECONDWRITE) and full support if memory regions are assigned recursive types through shape analysis (DDT, ARTISTE, MEMPICK).

Of the 8 works that support recursive types, 6 are dynamic and 2 are static approaches. The two static approaches (RHK, SECONDWRITE) recover only recursive pointers. While there has been a wealth of prior work on static shape analysis [Chase et al. 1990; Ghiya and Hendren 1996; Sagiv et al. 1999; Manevich et al. 2005; Berdine et al. 2007; Marron et al. 2009], those techniques have been applied to only source code so far. The process for identifying recursive types using dynamic approaches has three high-level steps: build memory graphs (already detailed in Section 4.6), identify memory graph regions corresponding to recursive types, and apply dynamic shape analysis to assign types to regions (e.g., linked list, binary tree).

Identifying regions in the memory graph. The high-level intuition behind partitioning a memory graph into regions (recursive types), first used in RDS, is that nodes in the same region should be of the same type and regions should be separated by nodes of different type. One challenge addressed only by MEMPICK is identifying overlapping data structures with nodes of the same type, for example, a tree in which leaf nodes form a linked list. DDT differs from other approaches in assuming that access to the recursive data structures happens through a small set of *interface functions*; thus, it misses them if they are accessed through inline manipulation, that is, without separate function calls.

Dynamic shape analysis. Types are assigned to memory regions by checking the shape of the regions against shape invariants. For example, a doubly-linked list node should have two pointers (*forward*, *back*) satisfying n→forward→back = n. The three approaches using dynamic shape analysis (DDT, ARTISTE, MEMPICK) output shapes of different granularity. All 3 identify cycles, lists (singly linked, doubly linked, circular singly linked, circular doubly linked), generic trees, and trees with parent pointer. In addition, DDT and MEMPICK analyze imbalance properties to refine trees, for example, balanced binary search trees, AVL trees, splay trees, and red-black trees. MEMPICK also identifies threaded trees.

5.6. Type Synonyms

$\boxed{\text{Column: } \textit{Type Synonyms}}$

Type synonyms are refinements of other types[2]. For example, in C, a programmer could define a type synonym for integers to represent currency amounts, sizes, zip codes, or unit life in a game (e.g., `typedef int Dollars`). Their representation is identical to the parent type; thus, they can only be distinguished from the parent type (and from other synonyms of the parent type) by how they are used. No work explicitly targets the recovery of type synonyms, but we have found two very related techniques, which we present here with a unifying view. Both techniques identify groups of variables used together by the program. Since all variables of the same type are not used together by the program, this essentially produces a partitioning of variables of the same type into type synonyms.

The first technique was introduced for source code by LACKWIT [O'Callahan and Jackson 1997] and applied to binary code (and byte code) in DYNCOMP [Guo et al. 2006]. It uses a flow-based unification approach in which the first time a variable is accessed by the program, a fresh type is created for it. The type is propagated similar to the primitive typing approaches, but with the difference that no type sources are defined. Thus, the type is never mapped to a known type, that is, the type system of the program is ignored. Variables that end up with the same type represent an abstract type that the programmer could (but may not) have used.

The second technique was introduced by REWARDS. It corresponds to assigning finer-grained types to the parameters or return values of external functions instead of the types in the source code or debugging symbols. For example, a call to the `open` Unix function returns an integer, but REWARDS ascribes it instead a more refined `file_d` type representing a file descriptor. Another possible approach would be propagating the *names* of the parameters instead of their types. For example, given the function `int cost(int miles, int price)`, we could assign the parameters the integer type synonyms `miles` and `price`. Thus, any variables using the parameters would be assigned those type synonyms instead of `int`.

5.7. Unions

$\boxed{\text{Column: } \textit{Unions}}$

Unions specify a number of member variables that they may store at the same memory location, but only one of those variables is stored at any given time. Untagged unions do not store an additional type tag to track the current member being used, and are supported only in weakly typed languages such as C/C++ and Cobol. Tagged

[2]Called abstract types in Guo et al. [2006]. They may also be called type abbreviations and type aliases.

unions are commonly used in functional languages (e.g., ML, Haskell) and can be seen as a record of a type tag and an untagged union.

Support for untagged unions requires handling multiple (possibly incompatible) types at the same location in the inferred type system. This can be done by representing a union as a record with multiple elements at the same (zero) offset (MYCROFT, IDA) or by defining a separate union type (TIE, ARTISTE).

A fundamental challenge with untagged unions is how to distinguish them from a typing conflict. While a sound analysis may not generate typing conflicts, none of the examined works makes such a strong soundness claim. In fact, approaches that solve constraints offline may use unions during constraint solving to resolve constraint violations. For example, MYCROFT uses a C union to represent the target of pointers that access variables of different size. Approaches that solve constraints online need to decide whether to keep a single type for a location, thus resolving incompatible types immediately to $\perp$, or to keep track of all incompatible types as part of a union type (compatible types are refined using the $\sqcup$ lattice operation). Given this fundamental challenge, we do not assign full union support to any approach in Table II, but only partial support to approaches that can represent unions.

5.8. Abstract Types

$\boxed{\text{Column: } \textit{Abstract types}}$

Abstract types specify an interface (i.e., a set of allowed operations) but not the representation [Dekker and Ververs 1994]. For example, a stack is a common abstract type with push/pop/top operations, which can be represented using a linked list or an array. Similarly, sets and maps can be represented using trees or arrays of linked lists. Interestingly, we did not find any work that infers abstract types other than DDT that partially focuses on this problem. This is likely because abstract types cannot be identified by their representation, for example, an array of linked lists is not always a map. Instead, identifying abstract types may require examining the set of operations performed on the data structure. This is somewhat similar to object recovery, which requires identifying the class methods. We believe that abstract types are a good candidate for future work, especially if there is a need for data structure reuse.

5.9. Function Types

$\boxed{\text{Column: } \textit{Function types}}$

A function type captures the prototype of a function, that is, the number, location, and type of parameters and return values. Recovering function prototypes (function types) is a fundamental problem in binary code reuse [Caballero et al. 2010] and for interprocedural static analysis. This problem differs from identifying a function's entry point and boundary, that is, the instructions belonging to the function, which we consider part of disassembly [Kruegel et al. 2004; Bao et al. 2014].

The challenges in function type recovery are that parameters and return values are not explicitly defined in the binary code, that their locations (e.g., how many parameters are passed on register and stack) are dictated by an unknown calling convention, and that callee-saved registers may look like function parameters. In Table II, we assign partial support to approaches that include function types in their recovered type system (e.g., MYCROFT, IDA, TIE) and full support to those (BCR, SECONDWRITE) that identify callee-saved registers and recover the prototype without assuming a calling convention.

The intuition for identifying function parameters and return values was first introduced in Zhang et al. [2007]. A parameter is any register or memory location read by a

Table III. Comparison of the Binary Analysis Platforms Used in the Implementation of Binary Code Type Inference Approaches

				Target Arch. & OS								**Release**		
				CISC		**RISC**				**OS**				
Platform	**IR**	**Static analysis**	**Dynamic analysis**	**x86**	**x86-64**	**ARM**	**MIPS**	**SPARC**	**Power PC**	**Windows / PE**	**Linux / ELF**	**Open source**	**Free binary**	**Commercial**
BAP [BAP 2011]	BIL	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗
BitBlaze [Bitblaze 2008]	VINE	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗
Boomerang [Boomerang 2004]	RTL	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗
CodeSurfer/x86 [CodeSurfer 2005]	CodeSurfer	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Dyninst [Dyninst 2009]	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
IDA [IDA 2005]	IDA	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
iDNA [Bhansali et al. 2006]	Nirvana	✗	✓	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗
LLVM [LLVM 2004]	LLVM IR	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
PIN [PIN 2005]	✗	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗
QEMU [Qemu 2006]	TCG	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
ROSE [Rose 2000]	SAGE III	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗
SecondWrite [SecondWrite 2013]	LLVM IR	✓	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
SmartDec [SmartDec 2011]	Formulae	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗
Udis86 [Udis 2009]	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗
Valgrind [Valgrind 2007]	VEX	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

Note: Platform information collected early October 2015.

function before being written, with some exceptions such as callee-saved registers and the stack pointer. Return values are locations written by a function and used by the program after the function returns. This definition captures not only the return value but also the function's side effects, for example, a memory buffer passed by reference that the function modifies.

Some limitations of this definition are that return values not used by the program cannot be identified, and unused parameters only in some situations (e.g., if placed between two other stack parameters). Some limitations of current approaches are that they do not recover complex parameters, for example, a pointer to a nested or recursive type, and that they do not support *variadic functions*, that is, functions that may take a variable number of arguments, such as `printf`.

6. IMPLEMENTATIONS

This section systematizes the implementation of the 38 approaches. Building a binary code type inference approach from scratch requires significant effort. Thus, 35 of 38 approaches build on top of previously available binary analysis platforms, which provide basic functionality such as disassembly and control flow graphs for static analysis and instruction-level monitoring for dynamic analysis. This section first describes the platforms used by the approaches (Table III), then the implementation of the approaches using those platforms (Table IV). Note that the implementation of an approach may use only a subset of the functionality offered by the underlying platform.

Platforms. Table III summarizes the 15 platforms used by the 38 approaches in Table III. The most common platform is BitBlaze, which provides both static and

Table IV. Comparison of the Implementations of Binary Code Type Inference Approaches

			Targeted Arch. & OS								Release		
			CISC		RISC				OS				
System	**Platforms**	**Uses IR**	**x86**	**x86-64**	**ARM**	**MIPS**	**SPARC**	**Power PC**	**Windows**	**Linux**	**Open source**	**Free binary**	**Commercial**
Mycroft [Mycroft 1999]	-	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
Ida [Guilfanov 2001]	IDA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
Ew [Emmerik and Waddington 2004]	Boomerang	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✗
Rds [Raman and August 2005]	PIN	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
x86sa [Christodorescu et al. 2005]	IDA	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
DynCompB [Guo et al. 2006]	Valgrind	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
Ffe [Lim et al. 2006]	CodeSurfer/x86	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Divine [Balakrishnan and Reps 2007]	CodeSurfer/x86	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Polyglot [Caballero et al. 2007]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Autoformat [Lin et al. 2008]	Valgrind	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Wckk [Wondracek et al. 2008]	-	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Laika [Cozzie et al. 2008]	-	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Tupni [Cui et al. 2008]	iDNA	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Dc [Dolgova and Chernov 2008]	SmartDec	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Dispatcher [Caballero et al. 2009]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Ddt [Jung and Clark 2009]	LLVM	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗
Bcr [Caballero et al. 2010]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Rewards [Lin et al. 2010]	PIN	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Ftc [Fokin et al. 2010]	IDA	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Dde [Slowinska et al. 2010]	QEMU	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Tda [Troshina et al. 2010]	SmartDec	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Tie [Lee et al. 2011]	BAP,PIN	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Howard [Slowinska et al. 2011]	QEMU	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
SmartDec [Fokin et al. 2011]	SmartDec	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗
Recall [Dewey and Giffin 2012]	IDA	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
PointerScope [Zhang et al. 2012]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Artiste [Caballero et al. 2012b]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Undangle [Caballero et al. 2012a]	BitBlaze	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
SecondWrite [ElWazeer et al. 2013]	SecondWrite	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
Rhk [Robbins et al. 2013]	Dyninst	✓	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
MemPick [Haller et al. 2013]	PIN	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Top [Zeng et al. 2013]	QEMU	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Ym [Yan and McCamant 2014]	BitBlaze	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
ObjDigger [Jin et al. 2014]	ROSE	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗
Lego [Srinivasan and Reps 2014]	PIN	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Yb [Yoo and Barua 2014]	SecondWrite	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
vfGuard [Prakash et al. 2015]	IDA,PIN	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
VTint [Zhang et al. 2015]	Udis86	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗

dynamic analysis support. However, 5 of 7 approaches using BitBlaze use only its TEMU [Yin and Song 2010] dynamic analysis component, built on top of QEMU [Bellard 2005]. Dde, Howard, and Top also build on top of QEMU. Among dynamic approaches, QEMU (including TEMU) is most popular (10 approaches), followed by PIN [Luk et al. 2005] (6), and Valgrind [Nethercote and Seward 2007] (2). Most popular among static approaches is IDA [IDA 2005] (5), followed by SmartDec [Fokin et al. 2011][3] (3), CodeSurfer/x86 [Balakrishnan et al. 2005] (2), and SecondWrite [ElWazeer et al. 2013] (2). Note that these numbers may not reflect overall popularity of a platform.

Table III shows the IR used by the platform, whether it supports static and dynamic analysis, the target architectures and operating systems it supports, and how it is released (open source, free binary, or sold commercially).

Binary code analysis can operate directly on a particular instruction set (e.g., x86, ARM) or convert the instruction set into an IR. Using an IR has two main advantages. First, it is easier to reuse the analysis as a component for other analyses and for applying it to different architectures (by adding front ends that convert another instruction set to the IR). Second, complex instruction sets such as x86/x86-64 with hundreds of instructions, each with side-effects such as implicit operands and conditional flags, can be converted into a smaller set of simpler constructs, making the analysis code simpler. Among the 15 platforms, 12 use an IR. Only SecondWrite reuses directly the IR of another platform (LLVM). However, IRs may influence each other, for example, VINE uses VEX as an intermediate step and BIL evolved from VINE.

Among the 15 platforms, 11 support static analysis and 7 dynamic analysis. The functionality provided by static analysis platforms varies widely. IDA and Udis86 are disassemblers, Boomerang and SmartDec are decompilers, and the rest (BAP, BitBlaze, CodeSurfer/x86, Dyninst, LLVM, ROSE, SecondWrite) offer diverse static analysis functionality such as disassembly, building control flow graphs and call graphs, IR simplifications, and data flow propagation. All dynamic analysis platforms can run unmodified binaries (i.e., no need to recompile or relink). QEMU is a whole-system emulator that can run a full guest OS (e.g., Windows) on a potentially different host OS (e.g., Linux). Dyninst, iDNA, PIN, and Valgrind execute an unmodified target program with customizable instrumentation (e.g., through binary rewriting) on the CPU of the host system. BAP and BitBlaze build their dynamic analysis components on top of PIN and QEMU.

We mark support for an architecture if the platform provides a front end for reading the architecture's machine code at the time of writing[4]. Note that Table III does not show any architectural support for LLVM because currently LLVM does not provide front ends for reading their machine code into the LLVM IR, though it provides back ends to write out machine code for all the architectures in Table III. Other platforms, such as ROSE and Secondwrite, can read machine code and output statements in the LLVM IR. Three platforms (BitBlaze, CodeSurfer/x86, and SecondWrite) support only x86, another 4 x86 and x86-64 (Dyninst, iDNA, SmartDec, Udis86), and another three (IDA, QEMU, and Valgrind) support all 6 architectures evaluated. Overall, 14 platforms support x86, 9 support x86-64, and 5 platforms support ARM, MIPS, SPARC, and PowerPC. We mark support for an OS if the platform can read executables for that OS (PE or ELF) at the time of writing. Udis86, a popular x86/x86-64 disassembler, has no OS support marked because it disassembles a string of machine code, rather than an executable.

Of the 15 platforms in Table III, 10 are open source, PIN is released as a free binary without source code, IDA and SecondWrite are commercial tools acquired in binary form[5], and CodeSurfer/x86 and iDNA are internal tools not released to the public.

Approach implementation. Table III summarizes the implementation of the approaches. For each approach, it shows the platforms on which it builds, the target

[3]Originally called TyDec.
[4]October 2015.
[5]IDA provides a free binary of the 2006 5.0 version.

architectures and operating systems that it supports, and how it is released. More than half of the approaches (20/38) use an IR, typically provided by the underlying platform. All static approaches use an IR, but a majority of dynamic approaches (18/22) do not (even if the underlying platform has an IR). Of those 18, 4 are value-based approaches that do not use information flow (RDS, LAIKA, MEMPICK, LEGO) for which the benefit of using an IR is unclear. The other 14 perform flow-based analysis without an IR. These approaches need to propagate flows only on the small set of move-like instructions (e.g., `mov`, `push`, `pop`) that propagate type constraints. Since they do not need to reason about other instructions, using an IR may be overkill for their goal. Another reason could be that the effort to learn the IR may be too high.

The fact that a platform has support for an architecture in Table III does not necessarily mean that an approach built on top of that platform supports that architecture, as support for that architecture may have been added to the platform later or the approach may not have targeted that architecture. For example, BAP currently supports x86, x86-64, and ARM, but TIE (built on top) supports only x86.

Of the 38 approaches, three are open source (EW, DYNCOMPB, SMARTDEC); one is released as a free binary (OBJDIGGER); two are commercial tools acquired in binary form (IDA, SECONDWRITE); and 32 approaches have not been released in any form (independent of the platform they build on being open source).

7. EVALUATIONS

In this section, we examine how the accuracy of binary code type inference approaches is evaluated, and systematize this in Table V. We analyze 2 dimensions: benchmarks (Section 7.1) and evaluation methodologies (Section 7.2).

7.1. Benchmarks

Three of the papers do not evaluate their techniques at all (MYCROFT, IDA, DC). However, IDA is a popular commercial platform whose capabilities have since been widely tested. Overall, the average number of benign programs tested is 18, the median 9, the standard deviation 28, and the maximum 112 (TOP). Only 31% of approaches are evaluated on at least 10 programs, but there has been an increase since 2013, which may indicate that techniques are becoming more robust, or a higher bar for acceptance. The programs most used for evaluation are the GNU Core Utilities (Coreutils), which comprise basic file, shell, and text manipulation utilities of Unix/Linux. Coreutils are used by 7 approaches. The SPEC CPU benchmark that contains computing-intensive programs for integer and floating point manipulation is used by 4 approaches, split between SPEC CPU 2006 (3) and SPEC CPU 2000 (1). The majority (24/38) of approaches evaluate their techniques on other, often less popular, benign programs of their choice.

Of the 8 approaches that are evaluated on malware, 6 use dynamic approaches (BCR, REWARDS, AUTOFORMAT, LAIKA, DISPATCHER, TOP) and the other two (x86SA, OBJDIGGER) static. The predominance of dynamic approaches can be explained by the widespread used of polymorphism (e.g., packing) across malware families.

7.2. Evaluation Methodologies

The first three columns in the methodology section of Table V capture the methodologies used to evaluate the accuracy of the binary code type inference; the last two columns capture the performance evaluation.

We have identified 3 categories of methodologies used to evaluate the accuracy of binary code type inference results. The first category comprises 15 approaches that do not perform quantitative evaluation of their type inference results (have ✗ in *Quantitative* column). These approaches can be split into 3 subcategories. As mentioned earlier, 3 approaches do not evaluate at all (MYCROFT, IDA, DC). Another 6 approaches compare

Table V. Comparison of the Evaluations of Binary Code Type Inference Approaches

	Benchmark					**Methodology**				
System	**SPEC**	**Coreutils**	**Other benign**	**# Benign programs**	**Malware**	**Quantitative**	**Fine-grained metrics**	**Cross-compared**	**Runtime overhead**	**Memory overhead**
MYCROFT [Mycroft 999]	✗	✗	✗	0	✗	✗	✗	✗	✗	✗
IDA [Guilfanov 2001]	✗	✗	✗	0	✗	✗	✗	✗	✗	✗
EW [Emmerik and Waddington 2004]	✗	✗	✓	1	✗	✗	✗	✗	✗	✗
RDS [Raman and August 2005]	✓	✗	✗	9	✗	✗	✗	✗	✓	✓
x86SA [Christodorescu et al. 2005]	✗	✓	✓	3	✓	✗	✗	✗	✗	✗
DYNCOMPB [Guo et al. 2006]	✗	✗	✓	6	✗	✓	✗	✗	✓	✓
FFE [Lim et al. 2006]	✗	✗	✓	3	✗	✗	✗	✗	✗	✗
DIVINE [Balakrishnan and Reps 2007]	✗	✗	✓	24	✗	✓	✗	✗	✓	✗
POLYGLOT [Caballero et al. 2007]	✗	✗	✓	11	✗	✗	✗	✗	✗	✗
AUTOFORMAT [Lin et al. 2008]	✗	✗	✓	7	✓	✓	✗	✗	✗	✗
WCKK [Wondracek et al. 2008]	✗	✗	✓	7	✗	✓	✗	✗	✗	✗
LAIKA [Cozzie et al. 2008]	✗	✗	✓	10	✓	✓	✗	✗	✓	✗
TUPNI [Cui et al. 2008]	✗	✗	✓	8	✗	✓	✗	✗	✓	✗
DC [Dolgova and Chernov 2008]	✗	✗	✗	0	✗	✗	✗	✗	✗	✗
DISPATCHER [Caballero et al. 2009]	✗	✗	✓	4	✓	✓	✗	✗	✗	✗
DDT [Jung and Clark 2009]	✗	✗	✓	11	✗	✓	✗	✗	✓	✗
BCR [Caballero et al. 2010]	✗	✗	✓	5	✓	✓	✗	✗	✓	✗
REWARDS [Lin et al. 2010]	✗	✓	✗	10	✓	✓	✗	✗	✓	✓
FTC [Fokin et al. 2010]	✗	✗	✓	3	✗	✓	✗	✗	✓	✗
DDE [Slowinska et al. 2010]	✗	✗	✓	9	✗	✓	✗	✗	✗	✗
TDA [Troshina et al. 2010]	✗	✗	✓	9	✗	✓	✗	✗	✗	✗
TIE [Lee et al. 2011]	✗	✓	✗	87	✗	✓	✓	✓	✗	✗
HOWARD [Slowinska et al. 2011]	✗	✗	✓	9	✗	✓	✗	✗	✓	✗
SMARTDEC [Fokin et al. 2011]	✗	✗	✓	5	✗	✓	✗	✗	✗	✗
RECALL [Dewey and Giffin 2012]	✗	✗	✓	2	✗	✗	✗	✗	✗	✗
POINTERSCOPE [Zhang et al. 2012]	✗	✗	✓	10	✗	✗	✗	✗	✓	✗
ARTISTE [Caballero et al. 2012b]	✗	✗	✓	5	✗	✓	✗	✓	✓	✗
UNDANGLE [Caballero et al. 2012a]	✗	✗	✓	8	✗	✗	✗	✗	✓	✗
SECONDWRITE [ElWazeer et al. 2013]	✓	✗	✗	18	✗	✓	✓	✓	✓	✗
RHK [Robbins et al. 2013]	✗	✓	✓	84	✗	✗	✗	✗	✓	✗
MEMPICK [Haller et al. 2013]	✗	✗	✓	26	✗	✓	✗	✗	✗	✗
TOP [Zeng et al. 2013]	✗	✓	✓	112	✓	✗	✗	✗	✓	✗
YM [Yan and McCamant 2014]	✗	✓	✗	106	✗	✓	✗	✗	✗	✗
OBJDIGGER [Jin et al. 2014]	✗	✗	✓	6	✓	✓	✗	✗	✗	✗
LEGO [Srinivasan and Reps 2014]	✗	✗	✓	10	✗	✓	✓	✗	✓	✗
YB [Yoo and Barua 2014]	✓	✗	✓	27	✗	✓	✗	✗	✗	✗
vfGUARD [Prakash et al. 2015]	✗	✗	✓	10	✗	✗	✗	✗	✗	✗
VTINT [Zhang et al. 2015]	✓	✓	✗	33	✗	✗	✗	✗	✓	✗

against the ground truth (i.e., source code, debugging symbols, protocol or file specification) only qualitatively (EW, x86SA, FFE, POLYGLOT, RECALL, RHK). The remaining 6 approaches perform application-specific evaluation. These approaches do not evaluate their type inference techniques, but instead evaluate their final application (RDS, POINTERSCOPE, UNDANGLE, TOP, vfGUARD, VTINT). For example, UNDANGLE does not evaluate its pointer inference technique against ground truth, but does quantify its dangling pointer detection, and TOP evaluates its results by recompiling the decompiled source code.

The second category comprises 19 approaches that perform coarse-grained quantitative evaluation of their type inference results against ground truth. These approaches have ✓ in the *Quantitative* column and ✗ in the *Fine-grained Metrics* column. These approaches use Boolean correctness metrics that count the number of (in)correctly typed variables. These metrics are coarse-grained because types can overlap each other or be nested (e.g., primitive types inside arrays), and a type can subtype another, for example, a `num32` compared with a ground truth of `int` could be considered incorrect or correct (but imprecise).

The third category comprises 3 approaches that use the following 4 finer-grained metrics for quantifying their inference (TIE, SECONDWRITE, LEGO). TIE proposes two evaluation metrics: *distance* and *conservativeness*. Distance addresses how to compare compatible (but different) types. The distance between two types is the number of levels between them in the primitive lattice if they are subtypes of each other, and the maximum lattice height otherwise. Conservativeness is based on the fact that TIE outputs a range of primitive types for a variable (upper and lower bound). It is a Boolean metric that is true if the type in the source code is between the inferred upper and lower bound types. However, the other approaches output a single primitive type instead and need to artificially create a type range to use this metric (e.g., SECONDWRITE). SECONDWRITE defines another metric to compare multilevel pointers since in TIE `int**` was equivalent to `int*`. This metric measures the ratio from the inferred pointer levels to the source pointer levels. LEGO proposes a metric to compare two class hierarchies that compares the sets of recovered methods and the inheritance relationships.

Cross-comparison. A separate categorization is whether the type inference results are cross-compared against prior approaches (TIE, ARTISTE, SECONDWRITE). This is complicated by the fact that most approaches are not open source (Section 6). TIE and ARTISTE compare against REWARDS, which was made available by its authors. TIE also compares against the HexRays commercial decompiler. In these cases, the comparison is done on Coreutils using TIE's distance and conservativeness metrics. The comparison with HexRays shows that TIE is more conservative (90% conservativeness compared to 45% on structural types), In terms of distance, TIE is 1.5 away from the original C type, about 200% better than HexRays [Lee et al. 2011]. SECONDWRITE compares with DIVINE and TIE using the same metrics, but on different benchmarks, likely because the SECONDWRITE authors did not have access to the other systems. Thus, the comparison is not fair as SECONDWRITE evaluates on SPEC CPU 2006, TIE on Coreutils, and DIVINE on a different benchmark.

Performance. The rightmost two columns of Table V capture whether the works evaluate the performance, in particular, the runtime and memory overhead. Accuracy seems to be considered more important, as there are 18 approaches that evaluate the result accuracy, but not the performance. Overall, 17 works measure the runtime overhead and 3 measure the memory overhead.

8. DISCUSSION

In this section, we discuss further insights obtained through our systematization, point out remaining problems, and share some thoughts on future research.

A common representation. Research on binary code type inference would benefit from a common IR that allowed a common type system for binary and source code (at least C/C++) and even for debugging symbol information. Such an IR would greatly help in evaluating the accuracy of the inferred types. Some IRs, such as BAP or VEX, have been designed to represent binary code. Others, such as LLVM or SAGE III, are intermediate levels between source and binary levels in a compiler and closer to this requirement. Of these, LLVM is quite popular, mature, and supports multiple architectures. For these reasons, it seems to be gaining traction as an IR for binary code as well. However, the type information LLVM includes is less rich than what exists at the source level. In addition, it is not clear how to accommodate some specificities of binary code type inference such as storing incomplete type information, handling conflicting types, extending the type system to accommodate new types, and capturing types that do not explicitly appear in the source code (e.g., dynamic arrays). We believe that extensions to popular IRs such as LLVM to handle these specificities, and the design of new IRs that handle them, would be important contributions to binary code type inference.

Evaluation metrics. Most works perform qualitative evaluation of their type inference results. A reason for this is the shortage of evaluation metrics. Only 3 works (TIE, SECONDWRITE, LEGO) propose metrics for performing fine-grained quantification of the type inference results. The proposed metrics cover only primitive types, multilevel pointers (e.g., char**), and class hierarchies. We believe that new metrics that are more fine-grained, cover other types (e.g., records, arrays, unions, function types), and support nested types should be considered an important contribution of binary code type inference works. An additional benefit is that well-established metrics may enable the comparison of different approaches despite the corresponding tools not being released.

Underinferred types. Our analysis reveals that some types have been targeted by little work, e.g., type synonyms, abstract data types, and function types. For example, recovering function types in which parameters are pointers to aggregate or recursive types is an interesting problem related to the recovery of abstract data types. Typing variadic functions requires identifying how the (variable) number of parameters is passed to the function, which can use a separate count parameter, a sentinel value at the end of the list, or a format string. Furthermore, recovery of format-string-like types is an open area for research.

Type refinement. Types can be decorated with Boolean predicates (and propositions with quantifiers), which constrain the set of values described by the type. Such *refinement types* can capture invariants of the underlying values such as a range of possible integer values (e.g., Booleans, C enumerations), bounds for array indices, or valid address ranges for pointers.

Other languages. Nearly all works surveyed focus on programs written in C/C++, but it would be interesting to test binary code type inference approaches against executables compiled from other language families such as functional programming languages (e.g., Haskell, OCaml) or logic programming languages. One challenge is that those languages provide a wealth of higher types. Another challenge is the different types of polymorphism at the source level. Some types of polymorphism, such as function overloading in C++, which produces different functions in the binary code, can already be handled. Others may be more challenging, such as parametric polymorphism, which allows one definition of each polymorphic function that executes the same code with arguments of different type.

Other architectures. The vast majority of approaches focus on the x86/x86-64 architectures, but other architectures are also widespread. For example, while ARM and

PowerPC are commonly used by embedded devices, and have been the subject of recent vulnerability finding approaches [Cui et al. 2013; Zaddach et al. 2014; Costin et al. 2014], no approaches evaluate yet on those architectures.

Source code techniques. Adapting source code techniques to binary code is a common trend that can still be pursued. For example, current approaches to recover recursive types from binary code use dynamic shape analysis (Section 5.5). Existing static shape analysis operates on source code [Chase et al. 1990; Berdine et al. 2007; Yahav and Sagiv 2008]. The application of static shape analysis would enhance the type coverage of static analysis platforms such as SecondWrite and BAP. The challenge will be scalability, as static shape analysis has proven expensive even on source code.

Handling obfuscated executables. The vast majority of approaches assume that the executable is not obfuscated. Some obfuscation techniques that focus on obfuscating the program code can be bypassed through dynamic analysis, but data structure obfuscations have also been proposed [Lin et al. 2009; Giuffrida et al. 2012]. Recent work examines how secure variable splitting and merging obfuscations are [Slowinska et al. 2014], but further work on binary code type inference on obfuscated code is needed.

Expanding dynamic analysis coverage. A shortcoming of dynamic analysis is the limited coverage. Despite this, we observe rather limited support in existing dynamic approaches for increasing coverage through combining results from multiple executions. Two possible reasons may be the high learning curve for multipath exploration engines and that many applications do not require typing the full program code, but only some data of interest (e.g., game units, protocol messages, key malware data structures). We believe that the combination of static analysis and dynamic analysis will allow more easily expanded coverage, specially for obfuscated executables.

Static analysis trade-offs. The authors of SecondWrite propose that static analysis approaches trade off soundness for increased performance, as long as the reduced soundness does not result in significant accuracy degradation. In their experiments, they achieve a 352x improvement over previous techniques with similar accuracy by using a flow- and context-insensitive points-to analysis with limited cardinality and number of iterations. Such trade-offs are an interesting avenue for future work.

9. CONCLUSION

Type inference on executables is a challenging problem in binary code analysis, which is required, or significantly benefits, many applications. In the last 16 years, a large amount of research has been carried out to infer the data and code types from executables. In this article, we have systematized binary code type inference by examining its most important dimensions: the applications that motivate its importance, the proposed approaches, the types that those approaches infer, the implementation of those approaches, and their evaluation. We have also discussed limitations and pointed to underdeveloped problems and open challenges.

REFERENCES

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1, 4:1–4:40.

Andrea Allievi. 2014. Understanding and Defeating Windows 8.1 Kernel Patch Protection. Retrieved March 9, 2016 from http://www.nosuchcon.org/talks/2014/D2_01_Andrea_Allievi_Win8.1_Patch_protections.pdf.

Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. Codesurfer/x86—A platform for analyzing X86 executables. In *Compiler Construction*.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in X86 executables. In *Compiler Construction*.

G. Balakrishnan and T. Reps. 2007. DIVINE: Discovering variables in executables. In *International Conference on Verification, Model Checking, and Abstract Interpretation*.

Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *USENIX Security Symposium*.

BAP 2011. Binary Analysis Platform. Retrieved March 9, 2016 from https://github.com/BinaryAnalysisPlatform.

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*.

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. 2007. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*.

Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *International Conference on Virtual Execution Environments*.

Bitblaze 2008. Bitblaze: Binary Analysis For Computer Security. Retrieved March 9, 2016 from http://bitblaze.cs.berkeley.edu/.

Boomerang 2004. Boomerang decompiler. Retrieved March 9, 2016 from http://boomerang.sourceforge.net/.

Peter T. Breuer and Jonathan P. Bowen. 1994. Decompilation: The enumeration of types and grammars. *ACM Transactions on Programming Languages and Systems* 16, 5, 1613–1647.

Elie Bursztein, Mike Hamburg, Jocelyn Lagarenn, and Dan Boneh. 2011. OpenConflict: Preventing real time map hacks in online games. In *IEEE Symposium on Security and Privacy*.

Juan Caballero, Gustavo Grieco, Mark Marron, Zhiqiang Lin, and David Urbina. 2012b. *ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions*. Technical Report TR-IMDEA-SW-2012-001. IMDEA Software Institute, Madrid, Spain.

Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012a. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis*.

Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. 2011. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*.

Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. 2010. Binary code extraction and interface identification for security applications. In *Network and Distributed System Security Symposium*.

Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Conference on Computer and Communications Security*.

Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*.

Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping kernel objects to enable systematic integrity checking. In *ACM Conference on Computer and Communications Security*.

David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Xi Chen, Asia Slowinska, and Herbert Bos. 2013. Who allocated my memory? Detecting custom memory allocators in C binaries. In *Working Conference on Reverse Engineering*.

Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. 2005. String analysis for X86 binaries. In *ACM Workshop on Program Analysis for Software Tools and Engineering*.

Cristina Cifuentes. 1994. *Reverse Compilation Techniques*. Ph.D. Dissertation. Queensland University of Technology, Brisbane, Australia.

Cristina Cifuentes and Mike Van Emmerik. 1999. Recovery of jump table case statements from binary code. In *International Workshop on Program Comprehension*.

CodeSurfer 2005. CodeSurfer. Retrieved March 9, 2016 from http://www.grammatech.com/research/technologies/codesurfer.

Andrei Costin, Jonas Zaddach, Aurelien Francillon, and Davide Balzarotti. 2014. A large scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*.

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. 2008. Digging for data structures. In *USENIX Symposium on Operating Systems Design and Implementation*.

Ang Cui, Michael Costello, and Salvatore J. Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. In *Network and Distributed System Security Symposium*.

Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *ACM Conference on Computer and Communications Security*.

R. Dekker and F. Ververs. 1994. Abstract data structure recognition. In *Knowledge-Based Software Engineering Conference*.

David Dewey and Jonathon T. Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code. In *Network and Distributed System Security Symposium*.

Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *IEEE Symposium on Security and Privacy*.

E. N. Dolgova and A. V. Chernov. 2009. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software* 35, 2, 105–119.

Katerina Dolgova and Alexander Chernov. 2008. Automatic type reconstruction in disassembled C programs. In *Working Conference on Reverse Engineering*.

Dyninst 2009. Dyninst: Putting the Performance in High Performance Computing. Retrieved March 9, 2016 from http://www.dyninst.org/.

Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Mike Van Emmerik and Trent Waddington. 2004. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering*.

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation*.

Alexander Fokin, Yegor Derevenets, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ decompilation. In *Working Conference on Reverse Engineering*.

Alexander Fokin, Katerina Troshina, and Alexander Chernov. 2010. Reconstruction of class hierarchies for decompilation of C++ programs. In *European Conference on Software Maintenance and Reengineering*.

Yangchun Fu and Zhiqiang Lin. 2012. Space traveling across VM: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In *IEEE Symposium on Security and Privacy*.

Tal Garfinkel and Mendel Rosenblum. 2003. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed Systems Security Symposium*.

Rakesh Ghiya and Laurie J. Hendren. 1996. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM SIGPLAN Symposium on Principles of Programming Languages*.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*.

Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. 2012. OS-Sommelier: Memory-only operating system fingerprinting in the cloud. In *ACM Symposium on Cloud Computing*.

Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. 2014. Multi-aspect, robust, and memory exclusive guest OS fingerprinting. *IEEE Transactions on Cloud Computing*.

I. Guilfanov. 2001. A simple type system for program reengineering. In *Working Conference on Reverse Engineering*.

Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. 2006. Dynamic inference of abstract types. In *International Symposium on Software Testing and Analysis*.

István Haller, Asia Slowinska, and Herbert Bos. 2013. MemPick: High-level data structure detection in C/C++ binaries. In *Working Conference on Reverse Engineering*.

Raymond J. Hookway and Mark A. Herdeg. 1997. DIGITAL FX!32: Combining emulation and binary translation. *Digital Tech. J.* 9, 1, 3–12.

IDA 2005. IDA. Retrieved March 9, 2016 from https://www.hex-rays.com/products/ida/.

Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries. In *ACM Workshop on Program Analysis for Software Tools and Engineering*.

Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *ACM Conference on Computer and Communications Security*.

Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *ACM Workshop on Program Protection and Reverse Engineering*.

Changhee Jung and Nathan Clark. 2009. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *IEEE/ACM International Symposium on Microarchitecture*.

Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. 2010. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *IEEE Symposium on Security and Privacy*.

Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*.

James R. Larus and Thomas Ball. 1994. Rewriting executable files to measure program behavior. *Software Practice and Experience* 2, 197–218.

JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*.

Junghee Lim, Thomas Reps, and Ben Liblit. 2006. Extracting output formats from executables. In *Working Conference on Reverse Engineering*.

Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*.

Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Network and Distributed System Security Symposium*.

Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SIGGRAPH: Brute force scanning of kernel data structure instances using graph-based signatures. In *Network and Distributed System Security Symposium*.

Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. 2009. Polymorphing software by randomizing data structure layout. In *SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment*.

Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium*.

LLVM 2004. The LLVM Compiler Infrastructure. Retrieved March 9, 2016 from http://llvm.org/.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Roman Manevich, Eran Yahav, G. Ramalingam, and Mooly Sagiv. 2005. Predicate abstraction and canonical abstraction for singly-linked lists. In *International Conference on Verification, Model Checking and Abstract Interpretation*.

Mark Marron, Deepak Kapur, and Manuel Hermenegildo. 2009. Identification of logically related heap regions. In *International Symposium on Memory Management*.

Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 3, 348–375.

Alan Mycroft. 1999. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming Languages and Systems*.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*.

Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*.

N. Petroni, A. Walters, T. Fraser, and W. Arbaugh. 2006. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation Journal* 3, 4, (December 2006).

PIN 2005. Pin - A Dynamic Binary Instrumentation Tool. Retrieved March 9, 2016 from https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Network and Distributed Systems Security Symposium*.

Qemu 2006. QEMU: An open source processor emulator. Retrieved March 9, 2016 from http://www.qemu.org/.

G. Ramalingam, John Field, and Frank Tip. 1999. Aggregate structure identification and its application to program analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages*.

Easwaran Raman and David I. August. 2005. Recursive data structure profiling. In *Workshop on Memory System Performance*.

Edward Robbins, Jacob M. Howe, and Andy King. 2013. Theory propagation and rational-trees. In *International Symposium on Principles and Practice of Declarative Programming*.

Rose 2000. ROSE compiler infrastructure. Retrieved March 9, 2016 from http://www.rosecompiler.org/.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric shape analysis via 3-valued logic. In *ACM SIGPLAN Symposium on Principles of Programming Languages*.

Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native X86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*.

Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*.

Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on Binary Translation*.

SecondWrite. 2013. SecondWrite. Retrieved March 9, 2016 from http://www.secondwrite.com/.

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*.

Gabriel M. Silberman and Kemal Ebcioglu. 1993. An architectural framework for supporting heterogeneous instruction-set architectures. *IEEE Computer* 26, 6, 39–56.

Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary translation. *Communications of the ACM* 36, 2, 69–81.

Asia Slowinska, Istvan Haller, Andrei Bacs, Silviu Baranga, and Herbert Bos. 2014. Data structure archaeology: Scrape away the dirt and glue back the pieces!. In *SIG SIDAR Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Asia Slowinska, Traian Stancescu, and Herbert Bos. 2010. DDE: Dynamic data structure excavation. In *ACM SIGCOMM Asia-Pacific Workshop on Systems*.

Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*.

Asia Slowinska, Traian Stancescu, and Herbert Bos. 2012. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX Annual Technical Conference*.

SmartDec. 2011. SmartDec decompiler. Retrieved March 9, 2016 from http://github.com/smartdec/smartdec.

Venkatesh Srinivasan and Thomas Reps. 2014. Recovery of class hierarchies and composition relationships from machine code. In *Compiler Construction*.

Katerina Troshina, Yegor Derevenets, and Alexander Chernov. 2010. Reconstruction of composite types for decompilation. In *IEEE Working Conference on Source Code Analysis and Manipulation*.

Udis. 2009. Udis86 Disassembler. Retrieved March 9, 2016 from https://github.com/vmt/udis86.

David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. 2014. SigPath: A memory graph based approach for program data introspection and modification. In *European Symposium on Research in Computer Security*.

Valgrind. 2007. Valgrind. Retrieved March 9, 2016 from http://valgrind.org/.

Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. 2014. Dynamic hooks: Hiding control flow changes within non-control data. In *USENIX Security Symposium*.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*.

Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. 2008. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*.

Eran Yahav and Mooly Sagiv. 2008. Verifying safety properties of concurrent heap-manipulating programs. *ACM Transactions on Programming Languages and Systems* 32, 5, 18:1–18:50.

Qiuchen Yan and Stephen McCamant. 2014. *Conservative Signed / Unsigned Type Inference for Binaries using Minimum Cut*. Technical Report 14-006. Department of Computer Science and Engineering, University of Minnesota.

Heng Yin, Zhenkai Liang, and Dawn Song. 2008. HookFinder: Identifying and understanding malware hooking behaviors. In *Network and Distributed System Security Symposium*.

Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution*. Technical Report UCB/EECS-2010-3. EECS Department, University of California, Berkeley.

Kyungjin Yoo and Rajeev Barua. 2014. Recovery of object oriented features from C++ binaries. In *Asia-Pacific Software Engineering Conference*.

Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium*.

Junyuan Zeng, Yangchun Fu, Kenneth Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation-resilient binary code reuse through trace-oriented programming. In *ACM Conference on Computer and Communications Security*.

Junyuan Zeng and Zhiqiang Lin. 2015. Towards automatic inference of kernel object semantics from binary code. In *International Symposium on Research in Attacks, Intrusions and Defenses*. Kyoto, Japan.

Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Defending virtual function tables' integrity. In *Network and Distributed Systems Security Symposium*.

Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*.

Jingbo Zhang, Rongcai Zhao, and Jianmin Pang. 2007. Parameter and return-value analysis of binary executables. In *Computer Software and Applications Conference*.

Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. 2012. Identifying and analysing pointer misuses for sophisticated memory-corruption exploit diagnosis. In *Network and Distributed System Security Symposium*.

Received April 2015; revised November 2015; accepted January 2016