Using Reduced Execution Flow Graph to Identify Library Functions in Binary Code

Jing Qiu, Xiaohong Su, and Peijun Ma

Abstract—Discontinuity and polymorphism of a library function create two challenges for library function identification, which is a key technique in reverse engineering. A new hybrid representation of dependence graph and control flow graph called Execution Flow Graph (EFG) is introduced to describe the semantics of binary code. Library function identification turns to be a subgraph isomorphism testing problem since the EFG of a library function instance is isomorphic to the sub-EFG of this library function. Subgraph isomorphism detection is time-consuming. Thus, we introduce a new representation called Reduced Execution Flow Graph (REFG) based on EFG to speed up the isomorphism testing. We have proved that EFGs are subgraph isomorphic as long as their corresponding REFGs are subgraph isomorphic. The high efficiency of the REFG approach in subgraph isomorphism detection comes from fewer nodes and edges in REFGs and new lossless filters for excluding the unmatched subgraphs before detection. Experimental results show that precisions of both the EFG and REFG approaches are higher than the state-of-the-art tool and the REFG approach sharply decreases the processing time of the EFG approach with consistent precision and recall.

Index Terms—Reverse engineering, static analysis, inline function, library function identification, subgraph isomorphism and graph mining

1 Introduction

Binary code analysis is crucial for some types of tasks where source code may not be available or appropriate, such as binary modification, binary translation, computer security and forensics. From an analyst's perspective binary code contains a wealth of information that can be retrieved through reverse engineering techniques including: disassembling, control flow analysis, function abstraction, call graph construction, software architecture recovery, and other techniques. Compared to source code, stripped binary code lacks explicit information such as boundaries of functions, which undoubtedly increases the difficulty of working with binary analysis.

The first step in understanding the data and control flow of an unknown binary code is *function identification*, which consists of two steps. First, the binary code is divided into functions. Then it is determined whether a function is a library function (full identification) or if a part of the function is an inline library function (inline identification). This is the foundation of other analysis in the reverse engineering field.

A *library function* is a function whose instruction sequence and semantics are known. If a function in binary is a library function, there is no need to reverse engineer it, as attempting to do so would waste time. Furthermore, many library functions are in binary code and contain instructions and structures that are difficult or impossible to reverse engineer. In this way, the disassembly code of a binary will be much more readable if calls to library functions use the functions' symbolic names.

Full identification is easier if the binary code is linked with dynamic link libraries or the debug information of the binary code is available. For example, in a PE file [1], the information of dynamic linked library functions is in the `.idata` section of the file and can be easily recovered with related import tables. Hence, the target of full identification is generally a stripped binary with statically linked libraries.

Unfortunately, inline identification is a big challenge in any binary code. The main difficulty is the lack of an evident difference between an inline library function and the rest of the code in a function. Therefore, much research has been done on full identification [2], [3], [4], [5], while no work has been done so far on inline identification to the best of our knowledge.

Currently, the most commonly used approach for full identification is to apply pattern matching, such as IDA Fast Library Identification and Recognition Technology (FLIRT) [3]. IDA uses the first 32 bytes of a function as the matching pattern. It offers a trade-off between speed and accuracy. However, if it is applied to inline identification, there are two challenges to overcome, as described below.

1) Bytes of an inline library function, within a function, may be discontinuous (see Fig. 1a). A compiler may change the order of some instructions for instruction alignment and pipelining.
2) An inline library function may not only be discontinuous, but may also have different byte sequences after compiler optimization. The bold codes in Fig. 1b are the same function using different registers.

In this paper, we propose a method for both full and inline identification to solve these two problems. We employ a representation called execution flow graph (EFG) [6], which extends the control flow graph (CFG), to

• *The authors are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China.*
E-mail: {topmint, sxh, ma}@hit.edu.cn.

Manuscript received 19 Jan. 2015; revised 9 Aug. 2015; accepted 11 Aug. 2015. Date of publication 18 Aug. 2015; date of current version 19 Feb. 2016.
Recommended for acceptance by Z. Hu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2470241

```
.text:6EED20E5  xor     eax, eax
.text:6EED20E7  add     ebp, 4
.text:6EED20EA  mov     edi, [ecx+4]
.text:6EED20ED  or      ecx, 0FFFFFFFFh
.text:6EED20F0  repne scasb
.text:6EED20F2  not     ecx
.text:6EED20F4  dec     ecx
```

(a) *strlen()* (in bold) is discontinuous

```
.text:004069F1  neg     ecx
.text:004069F3  push    edx
.text:004069F4  lea     edx, [esp+2Ch+var_C]
.text:004069F8  sbb     ecx, ecx
.text:004069FA  push    0
.text:004069FC  and     ecx, edx
```

```
.text:00406A02  neg     edi
.text:00406A04  push    ecx
.text:00406A05  lea     ecx, [esp+34h+var_10]
.text:00406A09  sbb     edi, edi
.text:00406A0B  and     edi, ecx
```

```
.text:00406A11  neg     esi
.text:00406A13  sbb     esi, esi
.text:00406A15  push    edi
.text:00406A16  and     esi, edx
```

(b) A library function having different byte sequences after compiler optimization

Fig. 1. Examples of two difficulties in library function identification.

represent the internal structure of a function.[1] No matter the order of the instructions, the data and control dependence among the instructions remain unchanged. The EFG provides a reliable representation of data and control dependence of a function, which lays a foundation for solving the first problem. We utilize instruction numbering techniques to extract the invariant features of an inline function with different variants to deal with the second problem.

The main steps of the proposed method are listed below (Fig. 2).

1) *EFG construction*. Initially *basic blocks* [8] are created and EFGs for each block are then constructed and labeled as local EFGs. A *basic block* is defined as a sequence of instructions with one entry point and one exit point. Then, the EFG of the function is constructed by analyzing the control dependence from exits to entries of local EFGs.
2) *Local EFG serialization*. Local EFG serialization translates a local EFG into an instruction sequence, which is equivalent to reschedule the instructions of a block with the semantics preserved. The instruction sequence is converted to an Identification (ID) sequence by replacing each instruction with its ID. The ID of an instruction represents the character value of the instruction. A local EFG corresponds to a unique ID sequence. Therefore, if the ID sequences of two local EFGs are not identical, the two local EFGs are not isomorphic. This property of ID sequences will be employed to filter unmatched EFGs.
3) *REFG construction*. The reduced execution flow graph (REFG), which is a variant of an EFG, is introduced to reduce the work load of subgraph isomorphism testings (see Section 4). Let EFG G_T and G_L be the EFG of target function f_T and library function f_L, respectively. G_L is subgraph isomorphic to G_T if an instance of f_L is in f_T. A local EFG of G_L with both predecessors and successors cannot be embedded in a local EFG of G_T. Thus this local EFG can be reduced to a single node and compared as a whole in subgraph isomorphism testing. If the REFGs of G_T and G_L are subgraph isomorphic, G_T and G_L are subgraph isomorphic also. We will prove this in Section 4.3.
4) *Library function identification*. Library identification is actually an EFG subgraph isomorphism testing problem. We design five filters to speed up the testing according to the properties of REFG. If a library function f_L is successfully matched and its code can be outlined, then the code of f_L in the target function is identified.

1. The execution dependence graph (EDG) in [6] is renamed to the execution flow graph in this paper to distinguish it from Zilles' EDG [7]. His EDG is built from an execution trace and its definition is totally distinct from ours.

This paper is an expanded and refined version of our conference paper [6]. It provides further research and improvements based on [6]. Specifically, new contents include: (a) local EFG serialization is introduced, (b) a new representation called reduced execution flow graph is introduced, and (c) construction, subgraph isomorphism testings and evaluations for REFGs are given.

The rest of this paper is organized as follows. The backgrounds of library functions and x86 instructions are given in Section 2. In Section 3, the definition and construction algorithms of the EFG are given and then the local EFG serialization is discussed. In Section 4, the REFG is introduced, and the subgraph isomorphism detection method using REFGs is discussed. The algorithms of library identification and the five filters are given in Section 5. Experiments and discussions are shown in Section 6. Limitations and improvements are given in Section 7. Section 8 describes related works and finally Section 9 states the summary.

2 Background

2.1 Library Functions

Library functions are implementations of behavior that has a well-defined interface by which the behavior is invoked. The value of a library function is the reuse of the behavior. When a program invokes a library function, it gains the behavior implemented inside that library function without having to implement that behavior itself.

Most program languages have standard library functions. Programmers can also create their own custom libraries. Most code used by modern applications is provided by system library functions (i.e., system APIs) which implement the majority of system services.

2.2 Static Library

Static linking refers to performing linking during the creation of an executable or another object file. It is usually done by a linker. A static library is a set of library functions to be statically linked. On Microsoft Windows, static libraries are stored in *.LIB files.

Library functions that are invoked in a program are sometimes statically linked and copied into the executable file. This process, and the resulting stand-alone file, is known as a static build of the program. A static build may not need any further relocation if virtual memory is used and no address space layout randomization is desired.

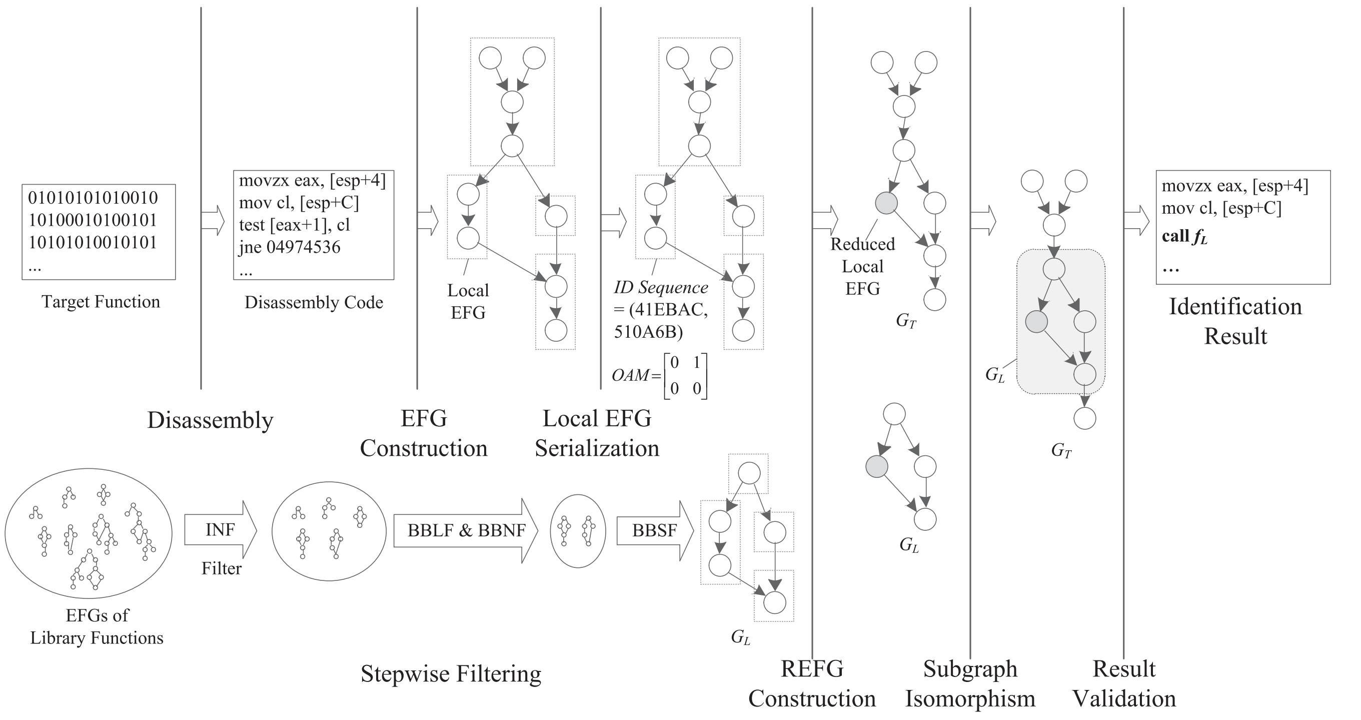


Fig. 2. An illustration of library function identification process.

2.3 Dynamic Library

Dynamic linking or late binding refers to linking performed while a program is being loaded (load time) or executed (run time), rather than when the executable file is created. A dynamically linked library is a library intended for dynamic linking. Only a minimum amount of work is done by the linker when the executable file is created; it only records what library routines the program needs and the index names or numbers of the routines in the library. The majority of the work of linking is done at the time the application is loaded (load time) or during execution (run time).

2.4 x86 Instructions in the Paper

Instruction "lea" computes the effective address of the source operand and stores it in the destination operand. The source operand is a memory address (offset part) specified with one of the processor's addressing modes; the destination operand is a general-purpose register. Instruction "xor" performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. Instruction "mov" copies the second operand (source operand) to the first operand (destination operand).

EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI are 32-bit general-purpose registers. The lowest 16-bits of these registers are AX, CX, DX, BX, SP, BP, SI, and DI. AL and AH are the low and high byte of AX, respectively. CL and CH are the low and high byte of CX, respectively.

3 Execution Flow Graph

3.1 Basic Definition

Let R_i/W_i denote registers or memory that instruction v_i reads or writes, and D_i denote the instruction address of v_i.

Definition 1 (Execution Flow). *There is an execution flow from instruction v_1 to instruction v_2 if and only if one of the following holds true:*

1) *C1: v_1 and v_2 are both in the same basic block, $(W_1 \cap R_2) \cup (R_1 \cap W_2) \cup (W_1 \cap W_2) \neq \emptyset$ and $D_1 < D_2$.*
2) *C2: v_1 and v_2 are both in the same basic block, and v_2 is a control transfer instruction (CTI).*
3) *C3: v_1 and v_2 are in two basic blocks B_1 and B_2, respectively. v_1 can be executed last in block B_1, and v_2 can be executed first in block B_2. B_2 is a successor of B_1 in the control flow graph.*

An execution flow from node v_1 to node v_2 is denoted by $v_1 \rightarrowtail v_2$.

C1 describes the following dependencies [9], [10]:

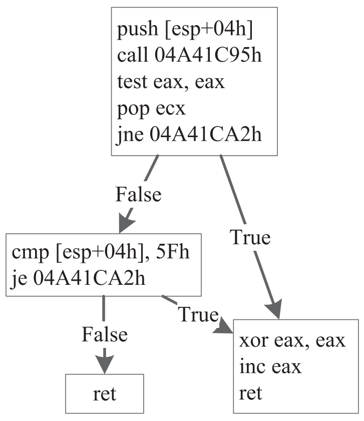
1) Flow (data) dependence: $W_1 \cap R_2$, v_1 writes something which will be read by v_2.
2) Anti-dependence: $R_1 \cap W_2$, v_1 reads something before v_2 overwrites it.
3) Output dependence: $W_1 \cap W_2$, v_1 and v_2 both write the same variable.

The condition $D_1 < D_2$ of C1 keeps the original execution order of the instructions.

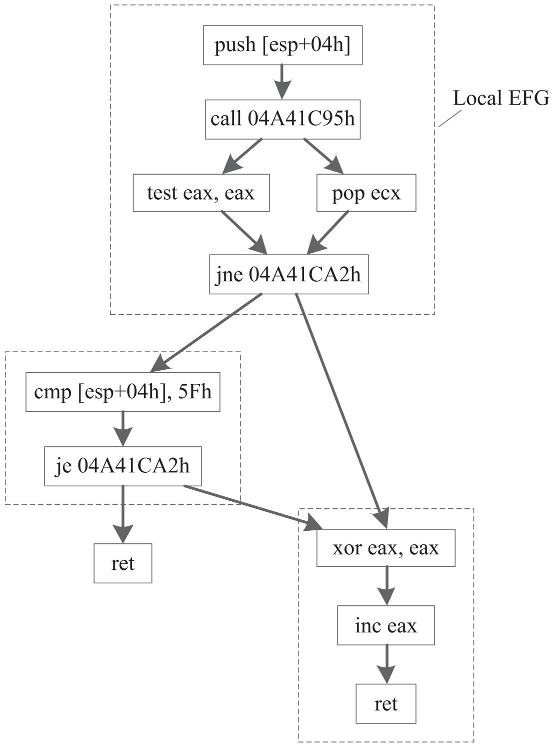
Definition 2 (Execution Flow Graph). *The execution flow graph G of a function P is a 3-tuple element $G = (V, E, \delta)$, where*

- *V is the set of instructions (i.e., nodes) in P.*
- *$E \subseteq V \times V$ is the set of execution flow edges.*
- *$\delta : V \rightarrow I$ is a function mapping from an instruction to a 32-bit integer.*

The implementation details of δ are described in Section 3.3.



(a) CFG



(b) EFG

Fig. 3. The CFG and EFG of *__iscsymf()* in VC7 LIBCMT.lib. Dotted boxes are local EFGs.

An EFG of a basic block is called a *local EFG*, and an EFG of a function is called a *global EFG* (Fig. 3). In the subsequent sections, an EFG is a global EFG unless specifically stated. In an EFG, a node without successors is called a *tail node*; a node without predecessors is called a *head node*.

An EFG is similar to a program dependence graph (PDG) [11], which is widely used in compiler optimization and source code static analysis [11], [12], [13]. The EFG of a basic block (local EFG) is actually the dependence graph of the basic block according to C1 and C2. The control-flow edges among basic blocks in an EFG are exactly the same as those in a basic-block level CFG. Thus, the EFG of a function (global EFG) is a hybrid representation of dependence graphs of basic blocks and the CFG of the function. As data dependence edges are confined to each basic block, the EFG of a function is much simpler than the PDG of the function.

Definition 3 (Graph Isomorphism for EFGs). *A bijective function $f: V \to V'$ is a graph isomorphism from an EFG $G = (V, E, \delta)$ to an EFG $G' = (V', E', \delta)$ if*

- *$\forall e = (v_1, v_2) \in E, \exists e' = (f(v_1), f(v_2)) \in E'$ such that $\delta(v_1) = \delta'(f(v_1))$ and $\delta(v_2) = \delta'(f(v_2))$.*
- *$\forall e' = (v'_1, v'_2) \in E', \exists e = (f^{-1}(v'_1), f^{-1}(v'_2)) \in E$ such that $\delta'(v'_1) = \delta(f^{-1}(v'_1))$ and $\delta'(v'_2) = \delta(f^{-1}(v_2))$.*

Definition 4 (Subgraph Isomorphism for EFGs). *An injective function $f: V \to V'$ is a subgraph isomorphism from G to G' if there exists a subgraph $S \subset G'$, such that f is a graph isomorphism from G to S.*

3.2 Disassembly and R/W Set

We obtained the operation information of the x86 instructions from BeaEngine [14], and analyzed which registers were read or written.

In order to avoid complicated inter-procedural analysis, the "call" instruction is assumed to read and write all variables. It is normally used to invoke a function while registers or memory variables are usually used to transfer arguments to a function.

For an instruction like "lea eax, [esi+18h]", the result is eax = esi + 18h. Therefore, "lea" is not treated as a memory accessing instruction, but as an assignment instruction. Instructions with prefixes "rep/repne", implicitly use "ecx". Thus, both R and W contain "ecx" in this case. In x86, some registers can be partially accessed. For example, "al" is a part of "eax". Accordingly, when "eax" is in R, "al" is also in R. Conversely, when "al" is in R, "eax" is in R also.

Meaningless instructions may be employed to align the address of the next instruction, such as "mov edi, edi", "nop", "lea esi, [esi]", etc. They are ignored in disassembly code.

3.3 Instruction Numbering

An inline library function may have various forms in target code (Fig. 1). For example, if an inline function uses a register as its parameter, the parameter may be different in different contexts. So if parameters can be generalized to a standard form, a single template can be used to identify all the variants. In our approach, instruction numbering is performed to convert an instruction to a standard form. Instruction numbering also makes instruction comparison and storage easier.

First, an instruction is normalized to eliminate the diversity of instructions through the following steps:

1) *Register normalization*. We use `reg1` to denote the first register in the operands of an instruction, and `reg2` to denote the second different register and so on, such as "mov eax, ecx" $\Rightarrow$ "mov reg1, reg2" and "xor eax, eax" $\Rightarrow$ "xor reg1, reg1".
2) *Memory normalization*. We use `M` to denote the memory accessing operand of an instruction, such as "mov eax, [ebx]" $\Rightarrow$ "mov reg1, M".

Second, the instruction after eliminating the diversity is represented as a two-tuple $SR = (m, r)$, where m represents the instruction mnemonic, and r represents the characteristic value of the operators. The digits in r, from high to low, denote the types of operands. For example, $r = 1$ for "op reg1", $r = 11$ for "op reg1, reg1", and $r = 21$ for "op reg1,

TABLE 1
Characteristic Values for an Instruction with 0, 1, or 2 Operands

Operand1	Operand2			
	NULL	Register	Memory	Immediate
NULL	0	10	40	50
Register	1	11 or 21	41	51
Memory	4	14	44	54
Immediate	5	15	45	55

reg2”. All characteristic values for an instruction with 0, 1 or 2 operands are listed in Table 1.

Finally, $SR = (m, r)$ is converted to a digit number called the ID of the instruction by invoking function $ID(SR)$. The function $ID(SR) = Hash(m) \mid (r \ll 16)$ is applied to obtain a 32-bit integer, which will be seen as the ID of the instruction. $Hash()$ is a string hash function that maps different instruction mnemonics to different 16-bit integers. Table 2 illustrates some examples of instruction numbering.

3.4 EFG Construction

We first partition the code of a function into basic blocks [8], then construct local EFGs for each basic block. Finally, for each local EFG, each tail node is connected to each head node of the successors of the local EFG.

Algorithm 1 shows the local EFG construction algorithm. In step 1, all possible dependencies in the basic block are checked, one by one. In line 3, j should be greater than i according to the definition of C1. In line 4, we determine whether the two instructions have data dependence.

Algorithm 1. Algorithm to Construct a Local EFG

```
Input: Basic block $B$
Output: Local EFG $G$
1  $L := B.Length - 1$;
   // Step 1: Create an initial graph
2  for $i := 0$ to $L$ do
3    for $j := i + 1$ to $L$ do
4      if $(R_i \cap W_j) \cup (W_i \cap R_j) \cap (W_i \cap W_j) \neq \emptyset$ then
5        $G := G \cup \{v_i \rightarrowtail v_j\}$;
     // Step 2: Simplify the graph
6  for $i := 0$ to $L$ do
7    for $j := i + 2$ to $L$ do
8      if $\{v_i \rightarrowtail v_j\} \in G$ then
9        for $k := i + 1$ to $j$ do //$i < k < j$
10         if $\{v_i \rightarrowtail v_k, v_k \rightarrowtail v_j\} \in G$ then
11           $G := G \backslash \{v_i \rightarrowtail v_j\}$;
```

After an initial EFG is created, it is simplified to an equivalent graph. The principle of simplification is based on the fact that for three different nodes v_i, v_j and v_k, if $v_i \rightarrowtail v_k, v_k \rightarrowtail v_j$, then v_i must be executed before v_j, therefore (v_i, v_j) is redundant and can be removed from the local EFG.

3.5 Local EFG Serialization

We proposed a method called *local EFG serialization* to quickly determine whether two local EFGs are isomorphic or not. The serialization transforms a local EFG to a node sequence, similar to re-scheduling the instructions. If only C1 and C2 in a basic block are considered, its local EFG is a directed acyclic graph. Therefore, topological sorting can be used in serialization. The result of topological sorting is that all nodes are sorted into a node sequence, where node u should precede node v if there exists $u \rightarrowtail v$.

The serialization process for local EFG G is as follows:

1) Choose the nodes whose incoming degree is 0 (without predecessors).
2) Sort and output the nodes chosen in step 1 in ascending order by their ID.
3) Remove nodes chosen in step 1 and their associated edges from G.
4) Repeat the above steps until no node remains in G.

Fig. 4 illustrates the calculation process for local EFG serialization.

Definition 5 (ID Sequence). *The ID sequence of a local EFG is a sequence of IDs, which is converted by replacing the nodes in a node sequence with their IDs.*

Theorem 1. *If ID sequences of two local EFGs are not identical, then the two local EFGs are not isomorphic.*

Proof. The ID sequence output from each iteration in the computation procedure is unique. As as result, the final ID sequence is also unique. Thus, if local EFGs are isomorphic, their ID sequences will definitely be the same. Therefore, if two ID sequences are not identical, their corresponding local EFGs will not be isomorphic. □

Thus, we can determine two local EFGs are not isomorphic by comparing their ID sequences. That is to say, it is true that two local EFGs are not isomorphic if their ID sequences are not identical, but not vice versa because the same ID sequence may correspond to multiple local EFGs as shown in Fig. 5. We cannot conclude that two local EFGs are isomorphic if their ID sequences are identical. So we introduce a new criterion, i.e., OAM.

Definition 6 (Ordered Adjacency Matrix, OAM). *Let v^i denote the ith node of a node sequence. The ordered adjacency matrix B of a local EFG $G = (V, E, \delta)$ on n nodes is the $n \times n$ 0-1 matrix where $B_{i,j} = 1$ iff $(v^i, v^j) \in E$.*

As shown in Fig. 5, two EFGs have different OAMs in spite of the same ID sequence. Thus, we can determine two

TABLE 2
Examples of Numbering Instructions

Instruction	Standardization	$SR = (m, r)$	$Hash(m)$	ID
mov eax, [esp+08h]	mov reg1, M	(“mov”, 41)	EBAC	41EBAC
push edi	push reg1	(“push”, 1)	C792	1C792
mov eax, ecx	mov reg1, reg2	(“mov”, 21)	EBAC	21EBAC

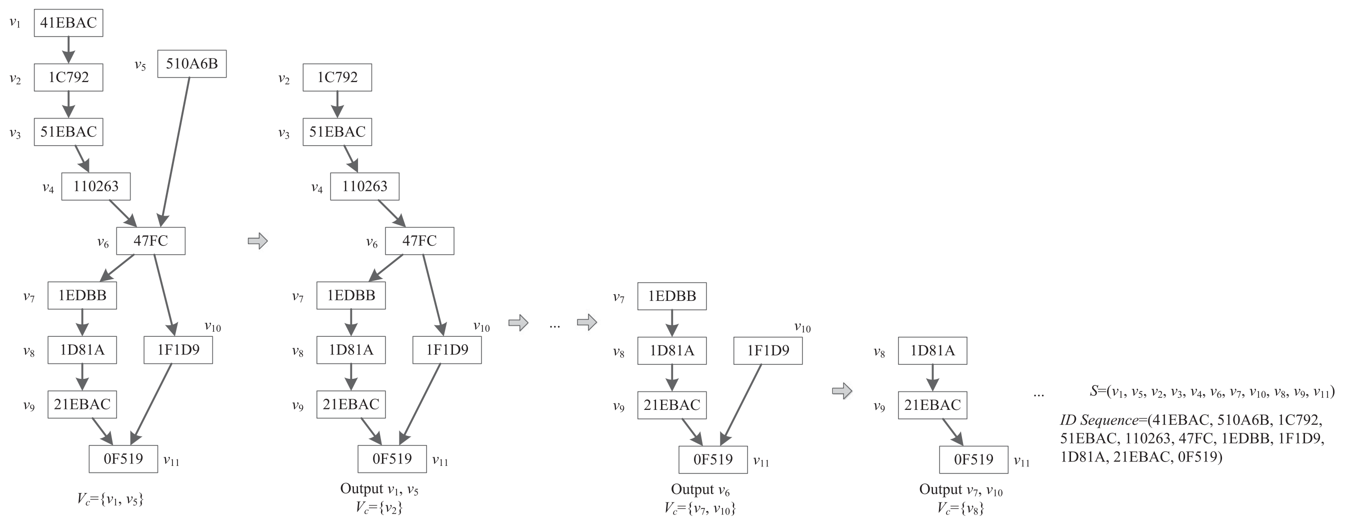


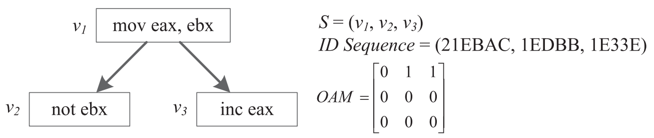
Fig. 4. An illustration of the calculation process for local EFG serialization. Here the IDs are marked in rectangles beside nodes. V_c is the set of chosen nodes in an iteration.

local EFGs are isomorphic if they have identical ID sequences and identical OAMs.

Theorem 2. *If both the ID sequences and OAMs of local EFG $G_1 = (V_1, E_1, \delta)$ and $G_2 = (V_2, E_2, \delta)$ are identical, then G_1 and G_2 are isomorphic.*

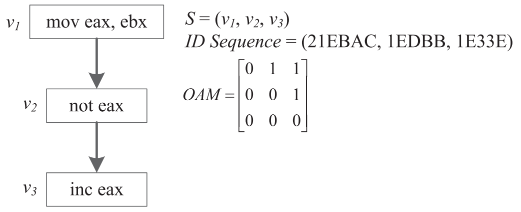
Proof. Let v_1^i and v_2^i denote the ith node in node sequences of G_1 and G_2, respectively. Identical OAMs indicate that if there exists $(v_1^i, v_1^j) \in E$, then there also exists $(v_2^i, v_2^j) \in E'$ and vice versa. Identical ID sequences indicate that for any ith node in node sequences of the two local EFGs, i.e., $\delta(v_1^i) = \delta(v_2^i)$. According to Definition 3, G_1 and G_2 are isomorphic. □

Two local EFGs are isomorphic if both the ID sequences and OAMs of the two local EFGs are identical. However, if two local EFGs are isomorphic, their ID sequences are identical while the OAMs are not always identical. Fig. 6 illustrates an example that isomorphic graphs may have different OAMs. The main reason is that v_2 and v_3 cannot be distinguished by their IDs in the sorting step of serialization as they have the same ID.



v_1 mov eax, ebx
v_2 not ebx
v_3 inc eax
$S = (v_1, v_2, v_3)$
ID Sequence = (21EBAC, 1EDBB, 1E33E)
$OAM = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

(a) A local EFG corresponds to a unique ID sequence



v_1 mov eax, ebx
v_2 not eax
v_3 inc eax
$S = (v_1, v_2, v_3)$
ID Sequence = (21EBAC, 1EDBB, 1E33E)
$OAM = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$

(b) Another EFG that corresponds to the same ID sequence

Fig. 5. Examples of EFGs in which an ID sequence may correspond to multiple EFGs. IDs of instruction "mov eax, ebx", "not ebx", "not eax", and "inc eax" are 0x21EBAC, 0x1EDBB, 0x1EDBB, and 0x1E33E, respectively.

For a graph, if all instructions in the sorting step of the serialization have different IDs, then both its ID sequence and OAM are unique.

4 Reduced Execution Flow Graph

4.1 Definitions

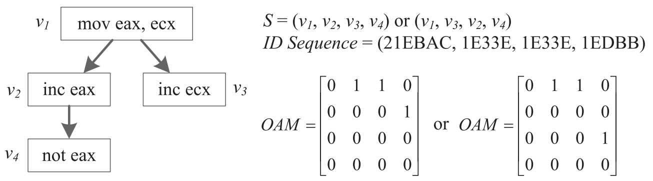
Definition 7 (Execution Unit). *An execution unit is an instruction or a local EFG.*

Definition 8 (Property of an Execution Unit). *For an instruction, its property is its ID. For a local EFG, its property is its ID sequence and OAM.*

Definition 9 (Reduced EFG, REFG). *The reduced execution flow graph G for a function F is a 3-tuple element $G = (V, E, \gamma)$, where*

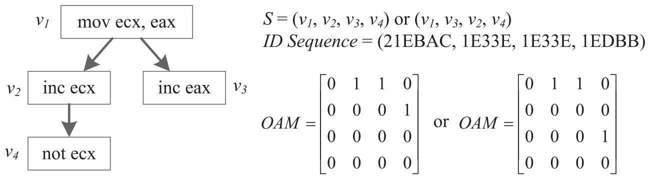
- *V is the set of execution units in F.*
- *$E \subseteq V \times V$ is the set of execution flow edges.*
- *γ is a function mapping from an execution unit to the property of the execution unit.*

Definition 10 (Graph Isomorphism for REFGs). *A bijective function $f : V \to V'$ is a REFG isomorphism from a REFG $G = (V, E, \gamma)$ to a REFG $G' = (V', E', \gamma)$ if*



v_1 mov eax, ecx
v_2 inc eax
v_3 inc ecx
v_4 not eax
$S = (v_1, v_2, v_3, v_4)$ or (v_1, v_3, v_2, v_4)
ID Sequence = (21EBAC, 1E33E, 1E33E, 1EDBB)
$OAM = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ or $OAM = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

(a)



v_1 mov ecx, eax
v_2 inc ecx
v_3 inc eax
v_4 not ecx
$S = (v_1, v_2, v_3, v_4)$ or (v_1, v_3, v_2, v_4)
ID Sequence = (21EBAC, 1E33E, 1E33E, 1EDBB)
$OAM = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ or $OAM = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

(b)

Fig. 6. An illustration of isomorphic graphs with same ID sequence but different OAMs.

- $\forall e = (v_1, v_2) \in E, \exists e' = (f(v_1), f(v_2)) \in E'$ *such that* $\gamma(v_1) = \gamma'(f(v_1))$ *and* $\gamma(v_2) = \gamma'(f(v_2))$.
- $\forall e' = (v'_1, v'_2) \in E', \exists e = (f^{-1}(v'_1), f^{-1}(v'_2)) \in E$ *such that* $\gamma'(v'_1) = \gamma(f^{-1}(v'_1))$ *and* $\gamma'(v'_2) = \gamma(f^{-1}(v'_2))$.

Definition 11 (Subgraph Isomorphism for REFGs). *An injective function* $f : V \rightarrow V'$ *is a graph isomorphism from REFG* G *to REFG* G' *if there exists a subgraph* $S \subset G'$, *such that* f *is a graph isomorphism from* G *to* S.

4.2 REFG Construction

Algorithm 2 shows the construction process of a REFG. First, all nodes and edges of G_1 are copied into G_2. Then the specified local EFGs in G_2 are replaced with execution units, and the related edges are adjusted.

Algorithm 2. EFG Reduction Function $reduce(G_1, B)$

```
Input: $G_1$: EFG
$B$: local EFGs to be reduced
Output: REFG
$G_2 := G_1$;
foreach Local EFG $b$ in $B$ do
  Replace $b$ with corresponding execution unit $v$;
  foreach Edge $(v_1, v_2)$ in $G_2$ do
    if $v_2$ is a head node in $b$ then
      $G_2 := G_2 \backslash \{(v_1, v_2)\} \cup \{(v_1, v)\}$;
    if $v_1$ is a tail node in $b$ then
      $G_2 := G_2 \backslash \{(v_1, v_2)\} \cup \{(v, v_2)\}$;
return $G_2$;
```

4.3 Subgraph Isomorphism Tests for EFGs Using REFGs

In this section, we will discuss how to use REFGs to test whether EFG G_2 is subgraph isomorphic to EFG G_1 or not. For convenience, a local EFG *contains* instruction v if and only if a node in the local EFG has the same ID with v.

In the next part, we will discuss in which cases a local EFG is reducible. To begin with, we define the following symbols.

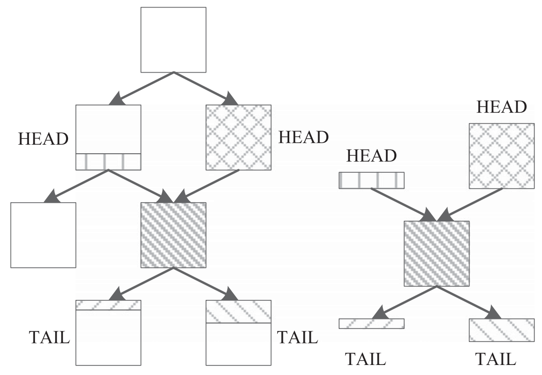
- Head block: A local EFG that has no predecessor.
- Tail block: A local EFG that has no successor and does not end with a CTI.
- Body blocks: The set of local EFGs that are not head blocks and tail blocks.
- $G.HT$: The head blocks and tail blocks in EFG G.
- $G.body$: The body blocks in EFG G.

The code of a local EFG that ends with a CTI cannot be a part of another basic block because no code can be inserted after the last instruction of this local EFG in the basic block. Therefore, this kind of local EFGs is reducible.

Fig. 7 indicates that only a head or tail block of G_2 can be a part of a local EFG of G_1. As a result, the following local EFGs are reducible:

- The body blocks of an EFG.
- Head blocks that are disconnected from any tail block.
- Tail blocks that are disconnected from any head block.

Suppose G_2 is the EFG of function f. All exit points of f are in tail blocks of G_2. There should be at least one path from a head block to a tail block. As a result, in G_1, a head block can be reduced if it is disconnected from any tail blocks. Similarly, in G_1, a tail block can be reduced if it is disconnected from any head blocks.



(a) G_1 (b) G_2

Fig. 7. Local EFGs of G_1 that contain all instructions of a head or tail block of G_2 are marked as a head or tail block.

Algorithm 3. REFG Construction $reduce\_for\_inline(G_1, G_2)$

```
Input: EFG $G_1$ and EFG $G_2$
Output: a REFG
$H := \emptyset, T := \emptyset, C := \emptyset$;
$found := false$;
//1. Assume all local EFGs in $G_1$ are reducible.
foreach Local EFG $b$ in $G_1$ do
  if $b$ has a unique node sequence then
    $C := C \cup \{b\}$;
//2. Identify head blocks of $G_1$.
foreach Local EFG $h$ in $G_2.head$ do
  foreach Local EFG $b$ in $G_1$ do
    if $b$ contain all instructions of $h$ then
      $found := true$;
      $H := H \cup \{b\}$;
if $|G_2.head| > 0$ and $!found$ then
  return NULL;
//3. Identify the tail blocks in $G_1$.
if $G_2$ has more than two local EFGs then
  $found := false$;
  foreach Local EFG $t$ in $G_2.tail$ do
    foreach Local EFG $b$ in $G_1$ do
      if $b$ contains all instructions of $t$ then
        $found := true$;
        $T := T \cup \{b\}$;
  if $|G_2.tail| > 0$ and !found then
    return NULL;
//4. Reduce isolated head and tail blocks in $G_1$.
if $G_2$ has head and tail blocks then
  $found := false$;
  foreach Local EFG $h$ in $H$ do
    foreach Local EFG $t$ in $T$ do
      if $h$ and $t$ are connected then
        $C := C \setminus \{h, t\}$;
        $found := true$;
  if $!found$ then
    return NULL;
return $reduce(G_1, C)$;
```

The REFG construction process is shown in Algorithm 3. First, all local EFGs of G_1 are marked to be reducible initially except the ones with uncertain node sequences, i.e., local EFGs that have two instructions with the same ID in the sorting step of serialization (Fig. 6). Only local EFGs that have unique node sequences can be tested for isomorphism by comparing their ID sequences and OAMs.

Second, the head blocks in G_1 are identified. A local EFG in G_1 is identified as a head block only if it contains all instructions of a head block in G_2. If G_2 has at least one head block but G_1 has none, G_2 cannot be subgraph isomorphic to G_1.

Third, the tail blocks in G_1 are identified. A local EFG in G_1 is identified as a tail block only if it contains all instructions of a tail block in G_2. If G_2 has at least one tail but G_1 has none, G_2 cannot be subgraph isomorphic to G_1.

Fourth, all possible checking pairs are checked. A checking pair consists of a head block and a tail block in G_1. The following cases should be considered:

1) G_2 has only one local EFG. It is unnecessary to check the connectivity.
2) G_2 has head blocks but no tail blocks or G_2 has tail blocks but no head blocks. It is unnecessary to check the connectivity as well because G_1 is in the same situation as G_2.
3) G_2 has both head blocks and tail blocks. If a head block is connected to a tail block in G_1 , both of them will be unmarked.

Finally, marked local EFGs in G_1 are input to function $reduce()$ to construct the REFG.

Theorem 3. *Let REFG G_1' = reduce_for_inline(G_1, G_2), and REFG G_2' = reduce_for_inline(G_2, G_2). If G_2' is subgraph isomorphic to G_1', G_2 is subgraph isomorphic to G_1 as well.*

Proof. Let f be a subgraph isomorphism from G_2' to G_1'. For each node v in G_2', $f(v) \in G_1'$:

1) If v is an instruction, $f(v)$ is also an instruction with the same ID as v. According to Algorithm 3, v is in G_2 and $f(v)$ is in G_1.
2) If v is an execution unit, $f(v)$ is also an execution unit with the same property as v. According to Theorem 2 and Definition 8, the two corresponding local EFGs in G_1 and G_2 are isomorphic.

For each edge (v_1, v_2) in G_2', $(f(v_1), f(v_2)) \in G_1'$:

1) If both v_1 and v_2 are instructions, both $f(v_1)$ and $f(v_2)$ are instructions. $(v_1, v_2) \in G_2$, $(f(v_1), f(v_2)) \in G_1$.
2) If v_1 is an instruction and v_2 is a reduced local EFG, $f(v_1)$ is an instruction and $f(v_2)$ is a reduced local EFG. The edge (v_1, v_2) in G_2' corresponds to the edge $(v_1, v_2.entry)$ in G_2. The edge $(f(v_1), f(v_2))$ in G_1' corresponds to the edge $(f(v_1), f(v_2).entry)$ in G_1.
3) If v_1 is reduced local EFG and v_2 is an instruction, $f(v_1)$ is a reduced local EFG and $f(v_2)$ is an instruction. The edge (v_1, v_2) in G_2' corresponds to the edge $(v_1.exit, v_2)$ in G_2. The edge $(f(v_1), f(v_2))$ in G_1' corresponds to the edge $(f(v_1).exit, f(v_2))$ in G_1.
4) If both v_1 and v_2 are reduced local EFGs, both $f(v_1)$ and $f(v_2)$ are reduced local EFGs. The edge (v_1, v_2) in G_2' corresponds to the edge $(v_1.exit, v_2.entry)$ in G_2. The edge $(f(v_1), f(v_2))$ in G_1' corresponds to the edge $(f(v_1).exit, f(v_2).\ entry)$ in G_1.

In conclusion, nodes and edges of G_2 are bijective with those of a subgraph of G_1. Thus, G_2 is subgraph isomorphic to G_1. □

Fig. 8 shows an example of subgraph isomorphism testing for EFGs by using REFGs. In Fig. 8, REFG $G_2' = reduce\_for\_inline(G_2, G_2)$ is subgraph isomorphic to REFG $G_1' = reduce\_for\_inline(G_1, G_2)$. EFG G_2 is subgraph isomorphic to EFG G_1. The first local EFG of G_2 is reduced because it is has a predecessor which is itself. The second local EFG of G_2 in which `sub eax, edx` is the only instruction is a tail block because it has no successor. Therefore, in G_1' local EFGs that does not contain `sub eax, edx` are reduced.

4.4 Graph Isomorphism Test for EFG Using REFG

In this section, we will discuss how to test that EFG G_2 is graph isomorphic to EFG G_1 using REFGs.

If G_1 is isomorphic to G_2, each local EFG of G_1 is graph isomorphic to a local EFG of G_2. All local EFGs with unique node sequences are reducible. They can be tested for isomorphism by comparing their ID sequences and OAMs.

Algorithm 4 shows the REFG construction for graph isomorphism testing.

Algorithm 4. REFG Construction $reduce\_for\_full(G)$

```
Input: EFG $G$
Output: REFG
$C := \emptyset$;
foreach Local EFG $b$ in $G$ do
  if $b$ has a unique node sequence then
    $C := C \cup \{b\}$;
return $reduce(G, C)$;
```

5 Library Function Identification

In this section, we will discuss whether the corresponding code of S can be outlined in different cases. A node in G_T rather than in S is defined as an *external node*; a node of S is defined as an *internal node*.

Next, we will discuss whether S can be outlined for one external node. Then S is an inline function if and only if S can be outlined for all external nodes. Algorithm 5 shows the validation algorithm.

For an external node v, if it can be executed before or after all nodes in S, then S can be outlined for v. There are two cases.

Case 1: v is a CTI (Fig. 9a). In this case, if the target of v is a node v' in S, then only if v' is executed first in S (i.e., v' has no predecessor in S), v can be executed before all nodes in S.

Case 2: v is not a CTI (Fig. 9b). The basic block that v belongs to should have no predecessor or no successor in S. Otherwise, in some execution path, the instructions in the preceding blocks should be executed before v or the instructions in the successor blocks should be executed after v. Thus, if and only if in the basic block v belongs to, v has no dependence on any node in S (v can be executed before all nodes in S), or no node in S has dependence on v (v can be executed after all nodes in S), S can be outlined.

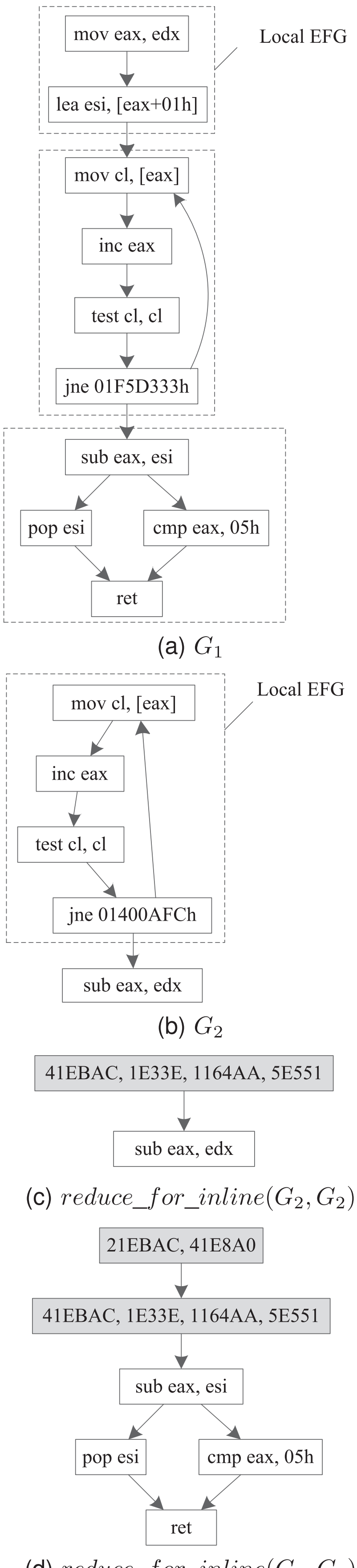


Fig. 8. An example of subgraph isomorphism testing for EFGs using REFGs. ID sequences are marked in shaded boxes which are execution units (i.e., reduced local EFGs).

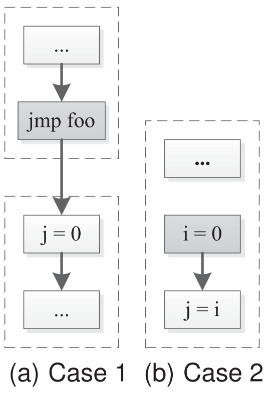


Fig. 9. Two cases for result validation where shadow nodes and dotted boxes represent external nodes and local EFGs, respectively.

Algorithm 5. Algorithm for Validating an Identification Result

```
Input: $G_T$: The EFG of $f_T$
$G_L$: The EFG of $f_L$
$S$: A subgraph of $G_T$ that is isomorphic to $G_L$
Output: true if $S$ can be outlined, false otherwise.
foreach Node $v$ in $G_T$ but not in $S$ do
  if $v$ is a CTI then //Case 1
    foreach Node $v' \in v.successor$ do
      if $v' \in S$ and $v'$ has a predecessor in $S$ then
        return false;
  else //Case 2
    $B$ := the basic block that contains $v$ ;
    if $B$ has no predecessor in $S$ then
      foreach Node $v' \in B$ do
        if $v' \in S$ and $v' \rightarrowtail v$ then
          return false;
    if $B$ has no successor in $S$ then
      foreach Node $v' \in B$ do
        if $v' \in S$ and $v \rightarrowtail v'$ then
          return false;
return true;
```

5.1 Filters

Subgraph isomorphism detection is an NP-complete problem [15] and is very time-consuming for large-scale graphs. However, some heuristics can help to reduce the complexity. In this study, we designed five filters where INF and HTF are first introduced to filter the impossible matches:

(i) *Instruction number filter (INF)*. Instructions of a function are classified into five categories: 1) Data transfer instruction, such as MOV, CMOV; 2) Arithmetic instruction, such as ADD, SUB; 3) Logical instruction, such as AND, OR; 4) String instruction, such as MOVS, LODS, STOS; 5) Other cases. If a category of f_T has fewer instructions than that of f_L, f_T and f_L cannot be matched.

(ii) *Basic block length filter (BBLF)*. A block length is the number of instructions of a basic block. Let V_1 and V_2 be the set of block lengths of f_T and f_L, respectively. If $\neg V_1 \cap V_2 \neq \emptyset$ (means that a block length is not in f_T, but is in f_L), the two functions cannot be matched.

(iii) *Basic block number filter (BBNF)*. If f_T has fewer basic blocks than f_L, they cannot be matched.

(vi) *Basic block sequence filter (BBSF)*. If a local EFG's ID sequence in f_L does not exist in f_T, then the two functions cannot be matched.

(v) *Head-tail filter (HTF)*. Suppose $G'_T = reduce\_for\_inline(G_T, G_L)$. If G'_T is empty, f_T and f_L cannot be matched. This filter is not shown in Fig. 2 as it is applied during the REFG construction. Heads and tails do not have to be identified in full identification, so HTF is only applied in inline identification.

Both the head and tail blocks of f_L are excluded in BBLF and BBSF because each of them might be a part of a basic block in a target function. INF, BBLF and BBNF is introduced in our previous work [6].

5.2 Library Identification Algorithm

The process for identifying a target function f_T with library functions F is as follows:

```
1) Disassemble $f_T$.
2) Select library functions $F'$ from $F$ that passes INF,
   i.e., $F' := \{f : f \in F \text{ and } f \text{ passes INF}\}$.
3) Build the CFG of $f_T$.
4) Select library functions $F_c := \{f : f \in F'$ and $f$ passes
   BBNF}.
5) For each library function $f_L$ in $F_c$:
   a) If $f_L$ does not pass BBLF, stop matching $f_L$.
   b) Build the EFG of $f_T$.
   c) If $f_L$ does not pass BBSF, stop matching $f_L$.
   d) Construct REFGs:
      • For inline identification:
          $G'_T = reduce\_for\_inline(G_T, G_L)$
          $G'_L = reduce\_for\_inline(G_L, G_L)$
      • For full identification:
          $G'_T = reduce\_for\_full(G_T)$
          $G'_L = reduce\_for\_full(G_L)$
   e) Detect all subgraphs of $G_T$ that are isomorphic to
      $G_L$.
   f) Check whether each detected subgraph can be
      outlined.
   g) Report each detected subgraph as a library
      function.
```

All EFGs of imported library functions are stored in a memory database via the FastDB technology [16] in our prototype system. INF is applied first because INF can be performed without control flow analysis. The numbers of instruction in each category and the number of blocks of a library function are indexed in the database for fast query. Because BBNF can be performed by one SQL query while BBLF cannot, BBNF is performed before BBLF. BBSF is performed subsequently. Finally, HTF is performed within the REFG construction function $reduce\_for\_inline()$.

6 Evaluation

We designed and implemented a prototype tool using C++. Source code is published at https://github.com/qiujing/libv. Test sets and online service can be accessed at http://sse.hit.edu.cn/fi. All EFGs are stored in a memory database via the FastDB technology [16]. Subgraph isomorphism testings are done by Aaron's parallel solution [17] which is an improved version of the VFlib2 algorithm [18] running on Cilk++ [19]. An IDA plugin is developed to export necessary information about the target functions from IDA. This information includes the start EA (Effective Address) and the end EA.

TABLE 3
Programs for Evaluation

Program	Version	Files	Compiler	Size/KB
7-Zip	9.2	9	VC7	3,386
Httpd	2.0.64	10	VC6	1,173
LAME	3.99.5	3	VC9	1,409
Notepad++	6.3.2	1	VC8	1,672
Putty	0.62	7	VC6	2,525
Python	3.3.1	9	VC10	3,596
WinMerge	2.14.0	4	VC9	2,950
PHP	5.4.14	39	VC9	15,794
WinXP	SP3	292	VC7	27,558

6.1 Setup

We evaluate our approach with the following binary sets:

- Eight open source applications including 7-zip 9.20, Apache HTTP Server 2.0.64, Notepad++ 6.3.2, WinMerge 2.14.0 and so forth. We compiled them with Microsoft Visual C++ with the /MT option, which is statically linked to the multi-thread C/C++ library. Program debug databases (PDBs) that hold debugging information are also generated for the ground truth.
- Binaries from the Windows XP SP3 operating system distribution. We obtained their PDBs from the Microsoft Symbol Server. These binaries are most likely compiled by VC7 (detected by ExeinfoPE [20] which is a packer detection tool).

The full Programs List is in Table 3. PHP was not compiled by us as its official site provides all PDBs. The evaluation was performed on an Intel i7-2600 machine with 8 GB memory and Windows 8 64 bit OS.

Library functions are as follows:

- String inline functions: strlen, strcpy, strncpy, strcmp, strncmp, strset, wcscat, wcsncat, wcscmp, wcsncmp, wcslen
- Compiler link libraries: libcmt.lib, libcpmt.lib in Microsoft Visual C++ 6 (from which we import 2,093 functions), 7 (3,963), 8 (6,688), 9 (7,035). Because identifying short functions will cause high false positive rates, a library function will be ignored if its byte size or instruction size is smaller than 6.

We chose IDA Pro 6.4 [21], which is widely considered the state-of-the-art tool, to compare with our approach. For efficient library function identification, our tool only processes libraries that are linked to the target. This does not mean that our approach is tied to a specific compiler version. In fact, we can always try all libraries of the evaluation, no matter how inefficient. In order to obtain results within a limited time frame, the time allotted for each subgraph

TABLE 4
Results of Identification

Program	Ours							IDA			
	INLINE		FULL					FULL			
	Found	Precision	True Positive	Precision	Recall	F-measure		True Positive	Precision	Recall	F-measure
7-Zip	40	1	**2,152**	0.75	**0.98**	0.85		1,843	**0.99**	0.84	**0.91**
Httpd	74	1	**1,570**	**1**	**0.94**	**0.97**		1,303	0.97	0.78	0.86
LAME	18	1	**1,178**	**1**	**0.94**	**0.97**		1,053	0.99	0.85	0.91
Notepad++	38	1	**420**	**0.98**	**0.77**	**0.86**		406	0.99	0.75	0.85
Putty	18	1	**1,687**	**1**	**0.96**	**0.98**		1,347	0.96	0.77	0.85
Python	24	1	**3,078**	**1**	**0.99**	**1.00**		1,562	0.97	0.78	0.86
WinMerge	9	1	**1,432**	0.98	**0.90**	**0.94**		647	**0.99**	0.74	0.85
PHP	173	1	**578**	0.99	**0.84**	**0.91**		517	0.91	0.75	0.82
WinXP	1,691	-	**1,816**	**0.99**	**0.36**	**0.53**		1,355	0.99	0.24	0.39

isomorphism testing is limited to a maximum of 10 seconds. A pair of EFGs is assumed to *not* be isomorphic if its subgraph isomorphism testing times out.[2]

The evaluation consists of two groups. In the first group, functions are identified by the EFG method. In the second group, functions are identified by the REFG method.

Reduction rate is defined to evaluate how much of an EFG is reduced

$$RR = 1 - \frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{M_i}\sum_{j=1}^{M_i}\frac{n'_j}{n_j}\right), \tag{1}$$

where

$M_i =$ number of subgraph isomorphism testing pairs of the ith file in the program,
$n_j =$ number of edges and nodes of the EFG of the target function in the jth testing pair,
$n'_j =$ number of edges and nodes of the REFG of the target function in the jth testing pair,
$N =$ number of files in the program.

We also use *precision*, *recall* and *F-measure* metrics to evaluate the effectiveness of an algorithm.

$$precision = \frac{TP}{R}, recall = \frac{TP}{GT},$$

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall},$$

where:

$TP =$ number of correctly identified functions,
$R =$ number of identified library functions,
$GT =$ number of library functions in ground truth.

An inline library function is part of a target function. Ground truth of inline functions is not available to us because it is not in PDBs. Therefore, if an inline function is identified in a target function, we review the corresponding code in the target function via IDA manually.

Functions *strlen()*, *strcpy()* and *strcmp()* are commonly used in C/C++ programs. In C++, operations of the standard string class implicitly contain these C string functions. For example, the copy constructor contains the semantic of *strcpy()*. Thus, it is normal that *strcpy()* is identified in a constructor of a string class. This kind of identification result is regarded as a true positive even if the identified inline function is not explicitly used in source code.

6.2 Results and Discussion

Table 4 gives the identification result. The results of the two groups are not given separately because they are identical. The recalls of inline identification are not computed because the ground truth of inline functions is not in PDBs. Time consumptions for inline identification, full identification are listed in Tables 5 and 6, respectively. Reduction rates are shown in Fig. 10 where INLINE and FULL represent inline identification and full identification, respectively.

6.2.1 Precision and Recall

Table 4 shows that our tool finds a greater quantity of full library functions in high precision and obtains higher recall than IDA, although the precision of our tool in 7-Zip is lower than that of IDA.

In our evaluation, the precision and recall of every program both in the EFG group and the REFG group are identical as expected. Compared to the EFG approach, the

TABLE 5
Time Consumption for Inline Identification

Program	EFG (sec)	REFG (sec)	REFG/EFG
7-Zip	3.78	0.19	5.03%
Httpd	1.75	0.05	2.86%
LAME	2.75	0.02	0.73%
Notepad++	1.42	0.02	1.41%
Putty	9.78	0.06	0.61%
Python	8.83	0.06	0.68%
WinMerge	1.53	0.09	5.88%
PHP	23.59	0.19	0.81%
WinXP	34.55	1.38	3.99%
Total	87.98	2.06	2.34%

2. In this paper, time-out functions that are given in our paper on SANER15 [6] are solved after using Aaron's parallel solution.

TABLE 6
Time Consumption for Full Identification

Program	EFG (sec)	REFG (sec)	REFG/EFG
7-Zip	1.55	0.97	62.58%
Httpd	0.44	0.33	75.00%
LAME	0.53	0.31	58.49%
Notepad++	0.34	0.19	55.88%
Putty	0.84	0.55	65.48%
Python	1.61	0.77	47.83%
WinMerge	0.81	0.41	50.62%
PHP	1.85	1.29	69.73%
WinXP	3.55	3.03	85.35%
Total	11.52	7.85	68.14%

efficiency of the REFG approach is significantly improved without any loss of precision and recall. The improvement of the identification speed comes from the high efficiency of subgraph isomorphism testing by comparing local EFGs as a whole and introducing HTF to prune most of the unmatched graphs before isomorphism testing.

The inline library functions are all manually checked. The checking results are proved to have no false positive identifications.

Two situations cause our tool to produce false positives for both the EFG and REFG method. First, two different functions may correspond to the same EFG. For instance, the function ~*CObjectVector()* which is not a library function in 7z.dll, is identified as ~*strstreambuf()* (Fig. 11). Moreover, identifying ~*CObjectVector()* causes large false positives in the identification of 7-Zip because class *CObjectVector* is heavily used in 7-Zip.

Second, incomplete EFGs of library functions prevent our tool from obtaining accurate identification results. On the one hand, our tool uses a linear sweep approach that disassembles the code of a function from the start address, one instruction after another. During the process of library function identification, our tool will stop disassembling when an

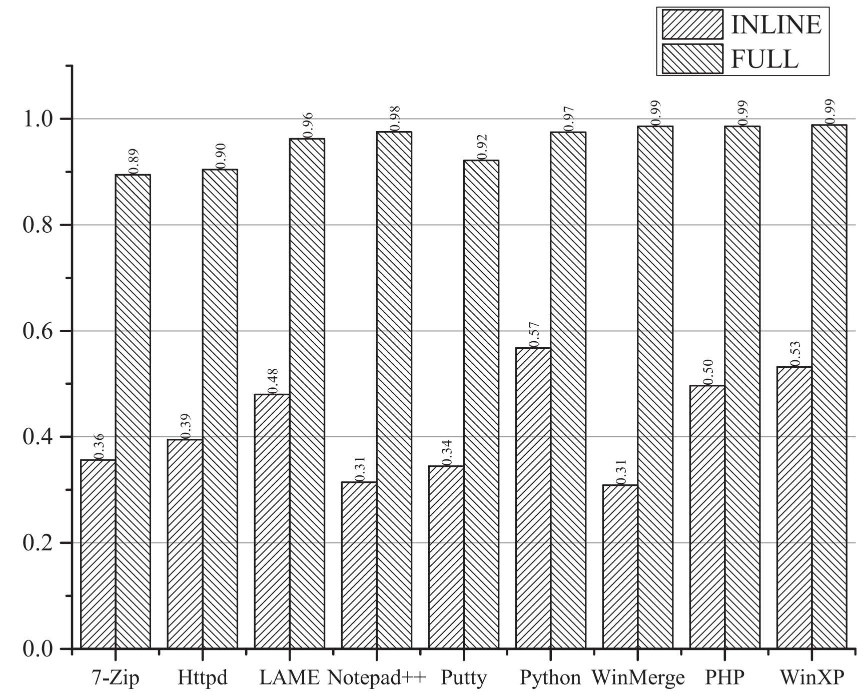
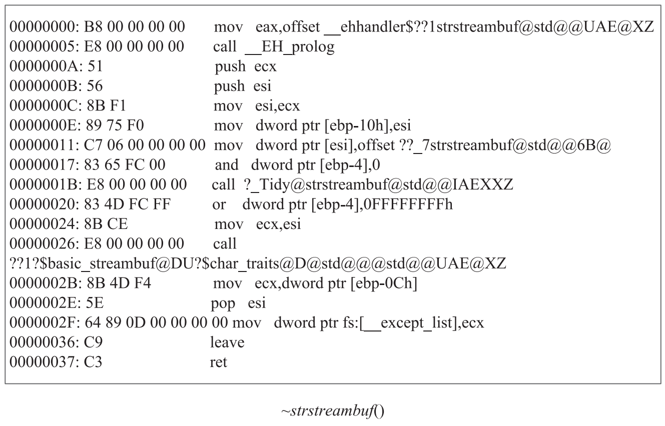
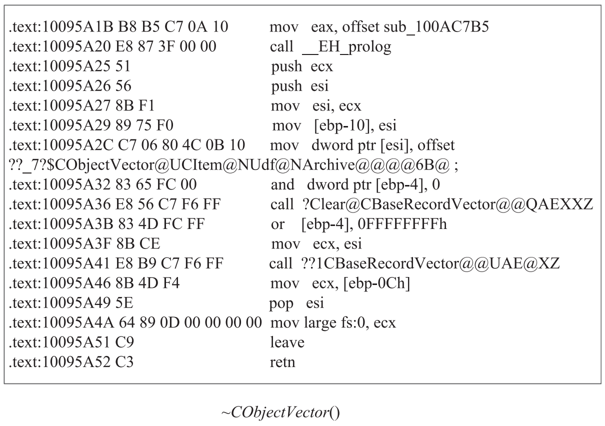


Fig. 10. Reduction rate for inline and full identification.



```
00000000: B8 00 00 00 00      mov   eax,offset __ehhandler$??1strstreambuf@std@@UAE@XZ
00000005: E8 00 00 00 00      call  __EH_prolog
0000000A: 51                  push  ecx
0000000B: 56                  push  esi
0000000C: 8B F1               mov   esi,ecx
0000000E: 89 75 F0            mov   dword ptr [ebp-10h],esi
00000011: C7 06 00 00 00 00  mov   dword ptr [esi],offset ??_7strstreambuf@std@@6B@
00000017: 83 65 FC 00         and   dword ptr [ebp-4],0
0000001B: E8 00 00 00 00      call  ?_Tidy@strstreambuf@std@@IAEXXZ
00000020: 83 4D FC FF         or    dword ptr [ebp-4],0FFFFFFFFh
00000024: 8B CE               mov   ecx,esi
00000026: E8 00 00 00 00      call
??1?$basic_streambuf@DU?$char_traits@D@std@@@std@@UAE@XZ
0000002B: 8B 4D F4            mov   ecx,dword ptr [ebp-0Ch]
0000002E: 5E                  pop   esi
0000002F: 64 89 0D 00 00 00 00 mov   dword ptr fs:[__except_list],ecx
00000036: C9                  leave
00000037: C3                  ret
```

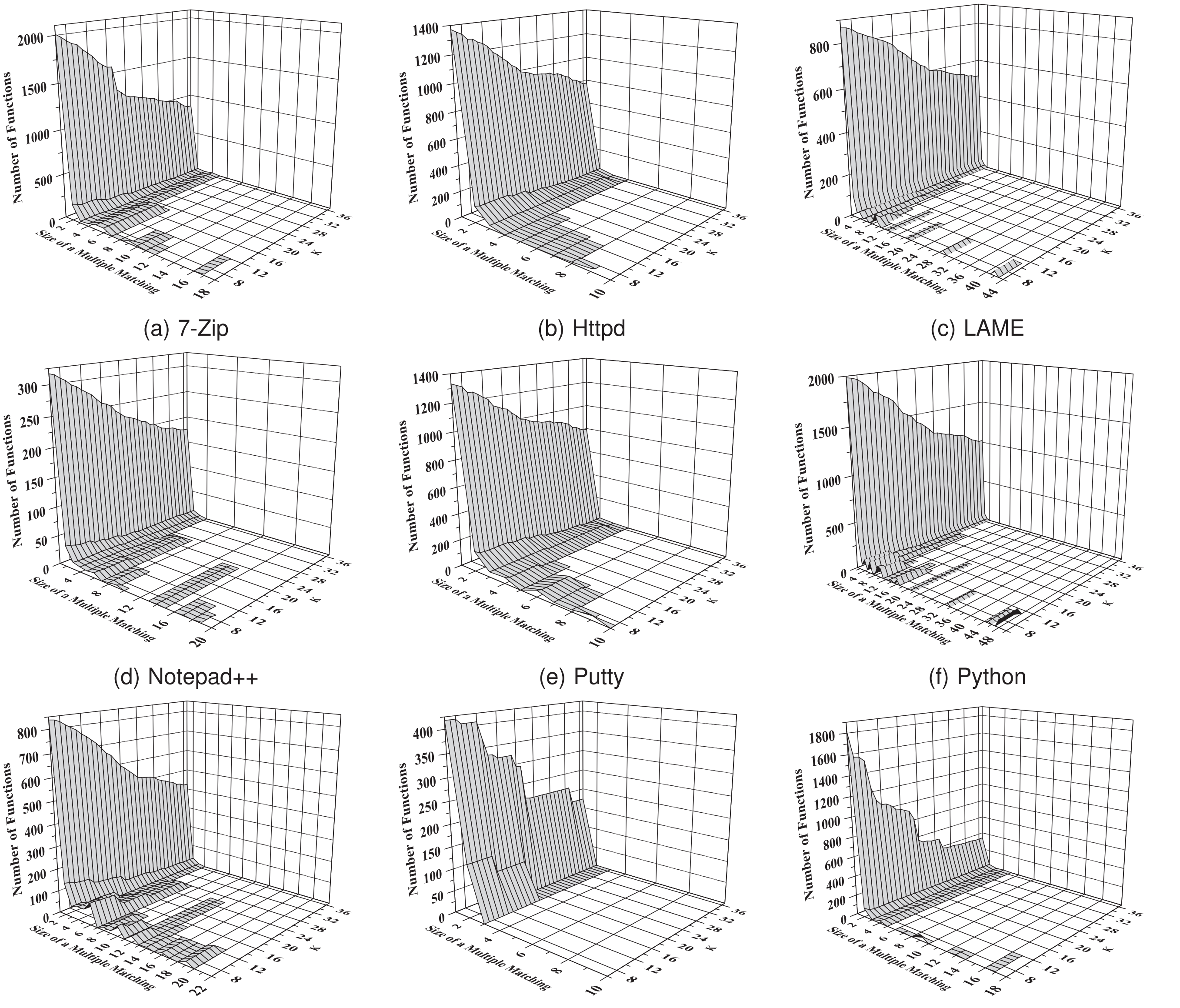
~*strstreambuf*()



```
.text:10095A1B B8 B5 C7 0A 10          mov   eax, offset sub_100AC7B5
.text:10095A20 E8 87 3F 00 00          call  __EH_prolog
.text:10095A25 51                      push  ecx
.text:10095A26 56                      push  esi
.text:10095A27 8B F1                   mov   esi, ecx
.text:10095A29 89 75 F0                mov   [ebp-10], esi
.text:10095A2C C7 06 80 4C 0B 10       mov   dword ptr [esi], offset
??_7?$CObjectVector@UCItem@NUdf@NArchive@@@@6B@ ;
.text:10095A32 83 65 FC 00             and   dword ptr [ebp-4], 0
.text:10095A36 E8 56 C7 F6 FF          call  ?Clear@CBaseRecordVector@@QAEXXZ
.text:10095A3B 83 4D FC FF             or    [ebp-4], 0FFFFFFFFh
.text:10095A3F 8B CE                   mov   ecx, esi
.text:10095A41 E8 B9 C7 F6 FF          call  ??1CBaseRecordVector@@UAE@XZ
.text:10095A46 8B 4D F4                mov   ecx, [ebp-0Ch]
.text:10095A49 5E                      pop   esi
.text:10095A4A 64 89 0D 00 00 00 00   mov large fs:0, ecx
.text:10095A51 C9                      leave
.text:10095A52 C3                      retn
```

~*CObjectVector*()

Fig. 11. ~*CObjectVector()* are misidentified as ~*strstreambuf()* by our tool.



(a) 7-Zip (b) Httpd (c) LAME

(d) Notepad++ (e) Putty (f) Python

(g) WinMerge (h) PHP (i) WinXP SP3

Fig. 12. Number of functions for an increasing K and an increasing number of multiple matchings. K denotes the minimum number of instructions of a library function. With the growth of K, our tool processes fewer and fewer library functions.

ambiguous jump or an undefined instruction is encountered. This strategy prevents it from producing too much assembly code which may cause false negatives. However, this strategy also leads to false positives, because two different functions may have the same bytes before the ambiguous jump.

Similarly, there are two situations that cause our tool to produce false negatives. First, incomplete EFGs of target functions may cause false negatives. Our tool assumes a function is an instruction sequence with both start and end addresses. Ranges of target functions that read from IDA are not always correct. Generally, the control flow leaves a function after a return instruction is executed. IDA additionally regards the instructions of library function calls that will terminate the program, e.g. `call exit()`, as a function's end. So the number of instructions of this target function in IDA is less than that of the corresponding library function. As a result, the library function will be filtered by INF. This will lead to false negatives.

Second, some library functions have completely different bytes than the target. For example, function *int __cdecl operator new(unsigned int size)* in 7z.dll and that of the imported link library have different bytes. Therefore, these functions are not identified by our tool.

In particular, our tool identified the functions that do not exist in PDBs and could not be identified by IDA, for example, $\_\_\_sbh\_alloc\_new\_region()$ in ipv6.exe of WinXP at 0x1009533.

6.2.2 Multiple Matching

For a target function, more than one library function may be matched. This is called a *multiple matching* and its size is the number of its candidates. For example, suppose a target function is identified as f_A, f_B and f_C due to their isomorphic EFGs. This result is a multiple matching and its size is 3. If the target function is actually one of these three functions, this result is regarded as correct. Fig. 12 shows that multiple matchings account for a very small proportion of the identification result. We use K to denote the minimum number of instructions of a library function. Both the number of true positives and the number of multiple matchings decrease with the growth of K. Most of the library functions of multiple matchings are C++ constructors and destructors.

6.2.3 Time Cost

No functions are time-out in inline identification because inline functions for the evaluation are generally much smaller (the longest one in this evaluation has 77 bytes and 28 instructions) than full functions.

In the evaluation, the functions identified by the REFG approach are the same as the EFG approach while the REFG approach takes only 9.96 percent total time of the EFG approach (2.34 percent for inline identification and 68.14 percent for full identification). This indicates that the REFG approach is more efficient than the EFG approach.

6.2.4 Reduction Rate

The reduction rate is defined to roughly indicate the degree of reduction of EFGs of target functions in the process of identification. According to Eq. (1), the higher reduction rate shows that the more proportion of instructions has been reduced to execution units.

Fig. 10 shows that reduction rates in inline identification of the program being evaluated are generally lower (about 42.16 percent) than that in full identification (about 95.45 percent). In full identification, G_T is expected to be graph isomorphic to G_L. Thus, all local EFGs of G_L and G_T are reducible. That is to say, many more local EFGs have a chance to be reduced in full identification than in inline identification.

In inline identification, G_T is expected to be subgraph isomorphic to G_L. All local EFGs of G_T that do not contain any instructions of any head and tail blocks of G_L need to be reduced. As most of inline functions are very short (the longest one in this evaluation has 77 bytes and 28 instructions), it seems that most local EFGs of a target function should be reduced. However, a head or a tail of an inline library function may contain a fewer number of instructions that may occur more frequently in a function. As a result, the reduction rate is lower than expectations. For example, the head of *strcpy*() has only one common instruction `xor eax, eax`. All local EFGs that contain this instruction are nonreducible.

6.2.5 Comparisons with IDA

IDA uses the first 32 bytes of a function as the feature of the function. IDA misidentifies the target function as the library function when only the first 32 bytes of a library function are the same as those of a target function but the rest of the bytes are different. For example, IDA misidentifies *_memmove()* as *_memcpy()* in 7-Zip.dll. Other examples include *_getc()* and *_fgetc()*, *__chkstk()* and *__alloca_probe()*, *__wsopen_s()* and *__sopen_s()*, etc. Our tool can identify these functions because it considers all bytes of a function.

Three factors make IDA produce false negatives. First, a library function may have different instruction sequences. In the modified binary sets of the evaluation, IDA fails to identify some library functions because these functions' first 32 bytes are different, such as the example in Fig. 1. Instruction normalization and EFG construction allow our approach to identify more semantically equivalent variations of a library function than IDA.

Second, some library functions such as *__adj_fdivr_m16i()* are not imported by IDA. In the disassembly code, IDA marks them as "unknown_libname_x" where x is an integer. Other examples include *___crtMessageBoxA()*, *__atodbl_l ()*, *__safe_fdiv()*, etc. However, these functions are in libcmt. lib of VC and they are imported into our tool. Thus, they can be identified by our tool.

Finally, some library functions such as *__onexitinit()* in an executable file are not identified as a function by IDA. This kind of function is not explicitly invoked by the instruction "call" and there is no cross reference to it. Our tool cannot identify these functions either because the addresses of target functions are exported from IDA.

Additionally, our tool can identify the functions that do not exist in PDBs and could not be identified by IDA, for example, *___sbh_alloc_new_region()* in ipv6.exe at 0x1009533.

7 Limitations and Improvements

There are several limitations of our approach. First, if the EFGs of different library functions are isomorphic, our approach cannot distinguish them. Two methods can improve the precision: 1) In the EFG of a library function, the vertex associated with the instruction `call` can attach relocation information as a property, and 2) the value of an immediate operand of an instruction should be preserved after instruction numbering.

Second, some compiler optimizations, such as common sub-expression elimination, may eliminate instructions of an inline function in the target code. As a result, the inline function within the target function is incomplete and may not be identified.

Third, the capacity of our approach is tied to library functions it processed. Traditional pattern-matching based techniques also have this limitation. We have utilized instruction standardization techniques to extract the internal template of an inline function. Similar forms of a given library function can be identified such as the library function in Fig. 1b. However, a library function may have completely different bytes in different versions. It is nearly impossible to provide all possible versions of a library function to our tool, although all available versions of this library function from known compilers can be provided easily. The implementation of a library function in an unknown compiler or in hand-crafted code may be totally different from that of known compilers. The root of this problem is that a computer cannot yet understand the meaning of the code of a function as well as a human can. It is worth noting that a recent work [5] has addressed this limitation. A new function representation called *semantic descriptor* is defined to represent the footprint of a library function. This function representation may be utilized to generalize a library function in various versions.

Finally, the analysis of memory-accessing instructions could be more precise. On one hand, avoiding alias analysis on memory operands causes our tool to produce false negatives. It is simple and crude that all memory accessing instructions are supposed to have data dependence on each other. In some situations, this hypothesis fails. For example, two adjacent instructions "A: mov [ecx], eax" and "B: inc [ecx+4]", are independent of each other. If the order of these instructions is A, B in a target function while it is B, A in the library function, the target function cannot be identified by our tool. To build more precise EFGs, pointer-alias analysis should be introduced. Pointer-alias analysis of binary code is difficult and its discussion is beyond the scope of this paper. On the other hand, in the process of numbering instructions, the abstraction of memory will lead to false positives. The current abstraction of a memory operand loses the details of the operand so that some instructions cannot be distinguished from each other by their IDs. For example, instruction "mov eax, [eax+4]" has the same ID as the instruction "mov eax, [eax]". This may lead to false positives.

Our future works include:

- Study better instruction numbering techniques in order to preserve more information from an instruction.
- Perform more accurate and extensive dependence analyses, such as pointer-alias analysis and inter-procedural analysis for function calls.

8 Related Work

8.1 Library Function Identification

In the past, various approaches to library function identification have been proposed. The well-known library identification technique is IDA FLIRT [3], which uses byte pattern matching algorithms to determine whether a target function matches any of the signatures known to IDA. Dcc [2] is similar to FLIRT. Hancock [4] extends the FLIRT technique to use library function reference heuristics, assuming that a function is a library function if the function is statically called within a library function. The main disadvantage of this type of method is that it only uses the first n bytes of a function as the matching pattern, and the precision can easily be improved by increasing n, but this method cannot ensure that all library functions are included. UNSTRIP [5] identifies wrapper functions in binary code based on their interaction with the system call interface. However, this approach is limited to wrapper functions. A library function may have no `call` instructions. So this approach is not suitable to identify these library functions.

GraphSlick is an IDA plugin that automatically detects inlined functions [22]. It starts with CFG construction and finds similar subgraphs in a program; However, it is more like a code clone detection tool and cannot provide function names of identified inline functions.

8.2 Code Mining

As a code mining problem, library identification is related to code clone detection, software plagiarism, and malware detection.

8.3 Source Level

At source level, a function can be represented by a program dependence graph, and clones can be identified by subgraph isomorphism detection [12], [23]. GPLAG is a PDG-based plagiarism detection tool that detects plagiarism by mining PDGs [24]. In GPLAG, graph isomorphism is used in plagiarism detection and a statistical lossy filter is proposed to prune the plagiarism search space.

PDG based approaches may not be appropriate for inline library identification for two reasons. First, data flow analysis for building a PDG is extremely time-consuming for large-scale functions. Second, for a function, the structure of the PDG is much more complicated than that of the EFG. Subgraph isomorphism is an NPC problem [15]. It may be impossible to finish the PDG subgraph isomorphism detection in an acceptable time. As a result, the implementation of GPLAG does not take control dependencies into account for efficiency concerns.

8.4 Binary Level

Sæbjørnsen et al. present a scalable algorithm for detecting code clones in binary executables [25]. Their algorithm extends an existing tree similarity framework based on clustering of characteristic vectors of labeled trees. Bruschi et al. [26] propose a strategy for the detection of metamorphic malicious code inside a program based on the comparison of the CFG of the program against the set of CFGs of known malware. Kruegel et al. [27] present a worm detection technique based on the analysis of a worm's CFG and introduces an original graph coloring technique that supports a more precise characterization of the worm's structure. These algorithms are in a "preprocess-normalization-and-compare" approach. They can be used in full identification but are not suitable for inline identification. They have not considered the data dependence among instructions of a block. As a result, a library function with discontinuous bytes cannot be identified by these approaches.

Recently, [28] presented a technique for searching binary code in executables. To compute similarity between functions, the technique decomposes them into continuous, short instruction sequences. Both our approach and the presented technique need to fight against compiler optimization. The main limitation of their approach is that it will produce bad results when matching small functions with less than 100 basic blocks.

Saed et al. proposed a new technique for identifying reused functions in binary code by matching traces (short pathes) of the Semantic Integrated Graph (SIG), which is a new representation that combines control flow graph, register flow graph, and function call graph [29]. Although both their and our approaches are based on graph representations, their matching procedure is totally different than ours. They define different types of traces such as normal traces, AND-traces, and OR-traces over SIG graphs, which are used for inexact matching. They carry out both exact and inexact matching between different binaries, where an exact matching applies to two SIG graphs with the same graph properties, whereas an inexact matching employs graph edit distance to measure the degree of similarity between two SIG graphs of different sizes.

9 Summary

It is difficult to identify the library functions with discontinuous byte sequences or variants by pattern matching. We presented a novel way to identify both full and inline library functions to solve this problem. Its novelty lies in that we convert library function identification to isomorphism identification of EFG subgraphs. Additionally, we introduced a new representation called reduced EFG in library function identification to reduce the time cost on subgraph isomorphism testing. We also developed five filters to prune the EFGs that cannot possibly be isomorphic to a target function. Experiments on some popular software by the prototype tool we implemented verified its effectiveness on speeding up subgraph isomorphism testing.

The main contributions of our work include:

1) Reduced Execution Flow Graphs (REFGs) introduced to speed up subgraph isomorphism testing for EFGs.
2) Two new lossless filters are designed to reduce the number of tests on subgraph isomorphism.
3) A fully working prototype is developed for evaluation.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful comments and suggestions. This research was funded by National Natural Science Foundation of China 61173021.

REFERENCES

[1] Microsoft. (2014). Microsoft pe and coff specification. [Online]. Available: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[2] M. V. Emmerik, "Signatures for library functions in executable files, Information Security Research Centre, Queensland University of Technology " Australia, Tech. Rep. FIT-TR-1994-02, 1994.

[3] I. Guilfanov, "Fast library identification and recognition technology," http://www.hex-rays.com/idapro/flirt.htm, 1997.

[4] K. Griffin, S. Schneider, X. Hu, and T. Chiueh, "Automatic generation of string signatures for malware detection," in *Proc. Recent Adv. Intrusion Detection*, 2009, pp. 101–120.

[5] E. R. Jacobson, N. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *Proc. 10th ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools*, 2011, pp. 1–8.

[6] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *Proc. IEEE 22nd Int. Conf. Softw. Anal., Evol. Reeng.*, 2015, pp. 261–270.

[7] C. B. Zilles, "Master/slave speculative parallelization and approximate code," Ph.D. dissertation, Computer Sciences Department, University of Wisconsin-Madison, 2002.

[8] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA, USA: Addison-Wesley, 2007, vol. 1009.

[9] D. L. Kuck, *Structure of Computers and Computations*. New York, NY, USA: Wiley, 1978.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 1981, pp. 207–218.

[11] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[12] J. Krinke, "Identifying similar code with program dependence graphs," in *Proc. 18th Working Conf. Reverse Eng.*, 2001, pp. 301–309.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[14] (2014). BeaEngine. [Online]. Available: http://www.beaengine.org/

[15] D. Johnson and M. Garey, "Computers and intractability: A guide to the theory of NP-completeness," San Francisco, CA, USA: Freeman, 1979.

[16] K. Knizhnik. (2014). Fastdb: Main memory database management system. [Online]. Available: http://www.garret.ru/fastdb.html

[17] A. Blankstein and M. Goldstein. (2009). Parallel subgraph isomorphism. [Online]. Available: http://courses.csail.mit.edu/6.884/spring10/projects/austein-blanks_subgraph_paper.pdf

[18] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "Performance evaluation of the vf graph matching algorithm," in *Proc. Int. Conf. Image Anal. Process.*, 1999, pp. 1172–1177.

[19] C. E. Leiserson, "The cilk++ concurrency platform," *The J. Supercomput.*, vol. 51, no. 3, pp. 244–257, 2010.

[20] (2014). Exeinfo pe. [Online]. Available: http://exeinfo.atwebpages.com

[21] (2014). The IDA Pro disassembler and debugger. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[22] (2014). Graphslick. [Online]. Available: https://www.blackhat.com/us-14/sponsored-sessions.html Microsoft

[23] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Static Anal.*, 2001, pp. 40–56.

[24] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 872–881.

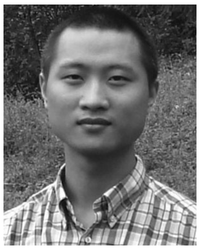
[25] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 117–128.

[26] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proc. Detection Intrusions Malware Vulnerability Assessment*, 2006, pp. 129–143.

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. Recent Adv. Intrusion Detection*, 2006, pp. 207–226.

[28] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 349–360.

[29] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.



Jing Qiu received the BS degree from the Harbin Institute of Technology (HIT) in 2005 and the MS degree from the Harbin University of Science and Technology in 2009. He is currently working toward the PhD degree from the HIT. His main interests include binary code analysis and binary code deobfuscation.

Xiaohong Su is a professor at the School of Computer Science and Technology, the Harbin Institute of Technology. Her main research interests include software bug detection, graphics and image processing, information fusion, and intelligent computation.

Peijun Ma is a professor at the School of Computer Science and Technology, the Harbin Institute of Technology. His research interest covers information fusion, software engineering, and pattern recognition.