GenLog: Accurate Log Template Discovery for Stripped X86 Binaries

Maosheng Zhang, Ying Zhao, Zengmingyu He
Department of Computer Science and Technology
Tsinghua University
Beijing, 100084, China
{zms.zsyyr, hsia.zachary}@gmail.com, yingz@tsinghua.edu.cn

***Abstract*—Computer systems often fail due to many reasons such as software bugs or administrator errors. Log analysis is an important method of computer fault diagnosis. With the ever increasing size and complexity of todays logs, the task of analyzing logs has become cumbersome to carry out manually. For this reason recent research has focused on the automatic analysis of these log files. But the system generated log files are always semi-structured, in some complex environment like computing center or cloud services, log files are more complicated. In order to get better effect in log analysis, log template extraction before analyzing is very important. However, the quality of log template is affected by the log file itself, some logs which have important pattern of event are hard to be extract just because they occur rarely in log files. On the other hand, source code of COTS (Commercial Off-The-Shelf) software or dynamic library files, such as DLL, are often not available. In view of the limitations of obtaining log template from log files, we provides a method which can extract log template from the stripped executables (i.e., neither source code nor debugging information need be available). We analyzing executables using static analysis techniques. By virtue of the advantages of slicing method from which we can get sets of log-generate-related instructions from executable, the problems can be simplified and analysis efficiency is improved remarkably. From this slicing part, we identify the hierarchies of log-generated-functions, rebuild the buffer in which log message is connect and refine it use a token-aimed trimming. So that we can get a complete log template set with complicated structure from executables with large size, especially system and net service binaries.**

I. Introduction

Logs play an important role in production computing systems as a means of recording operational status, environment changes, configuration modifications, errors encountered, and so on. For this reason, logs become an important source of information about the health or operation status of an application, a computing system, or infrastructure, and are relied on by system, network, or security analysts as a major source of information. As the size of logs has continued to grow with the ever-increasing size of todays computing infrastructure and softwares, the processing of reviewing event logs has become both difficult and error prone to be handled manually. Therefore automatic analysis of logs has become an important research problem that has received considerable attentions [2], [16]–[20], [25], [26], [34], [37], [38].

There are two chanllenges for automatic analysis of logs: (i) Understanding the semantic meaning of each log that normally records a timestamp, a source, and a message, which can be formatted into a log template; (ii) Understanding the call point in the program of each log, which is essential for failure diagnosis based on run-time logs.

The existing works on lot template discovery mainly focus on how to extract formats from logs generated by the program using data mining techniques such as finding frequent event type patterns [33] and classifying event types [19]. In addition to the chanllenges on efficiency posed by the tremendous size of production log files, these approaches often are sensitive to model tuning and fail to produce accurate log templates. For example, let us consider a log file with four log messages as below:

Connection 1 is OK
Connection 3 is OK
Connection 4 is INTERRUPT
Connection 5 is INTERRUPT

If we require any frequent patterns must be supported by at least two logs, we can get two log templates: `"Connection * is OK"` and `"Connection * is INTERRUPT"`. However, if we change the requirment to be supported by three logs, we can only get one template: `"Connection * is *"`. Especially, if a log record occurs rarely in a log file, it will not be extracted by such method, meanwhile it maybe is a key indicator of a system failure.

Another rich source for log analysis is the source code that produces logs [27], [28], [38]. However, for most Commercial-off-the-shelf softwares and library binary files, we could not get the source codes. Even for open source softwares, different versions may lead to a change of log generation infrastructure, so the log file has to match the exact version of the software. Furthermore, path inference tools based on run-time logs, such as SherLog [38] analyze the call point of each log on the fly which is time comsuming and not applicable to large application.

In the paper, we propose GenLog, a method to extract log templates and their call points from stripped X86 binaries. The merits of GenLog are two folds. (i) GenLog is able to produce accurate log templates only based on X86 executables, which is normally easier to get than source codes. (ii) GenLog is able

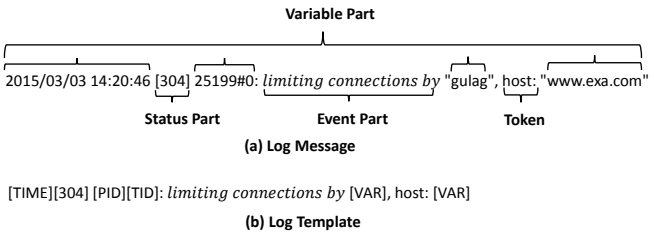

Fig. 1. Log Message and Log Template.

to produce call points in the program for each log template, which can be used to reduce the search space significantly by path inference tools, such as SherLog. GenLog employs an overall analysis of binary codes with static analysis techniques, and is able to handle the challenges posed by binaries, such as the lack of variable type information, virtual function pointers, and alias confusions.

To summarize, we make the following contributions.

1) We define the characteristics of logging related functions and define a complete path that ensures a log template includes its call points, its core construction function, and its output function. GenLog finds the complete paths of all log templates in a binary through a combined bottom-up and top-down slicing technique.
2) GenLog reconstructs the memory buffer in which log messages were constructed via date flow analysis, identifies components of log messages using taint propagation analysis, refines confusion parameters and call back functions using token-aimed trimming to extract the log template.
3) We report experimental results of GenLog on four X86 executables, one of them is a web service sofeware. The experiments show that GenLog can identify 99.9% log templates present in the testing log files correctly.

The structure of this paper is organized as follows. We first formulate the problem in Section II. GenLog is proposed in Section III. Experimental results are reported in Section IV. The related works are discussed in Section V and we conclude the paper in Section VI.

II. Problem Defintion

In this section we first define *log message* and *log template* formally, then we introduce the types of functions in binary codes that are related to log generation and define the *complete path* of these functions to identify log templates.

A. Log Message and Log Template

Definition 1 (Log Message): A text-based statement representing events that occur within the system or application processes on a computer system.

Log messages usually are generated with certain formats and have different components representing various aspects of an event. Fig. 1(a) shows an example of a typical log message in a web service system with the following components:

TABLE I
Tag of Variable Term

Tag	Function
TIME	GetSystemTime
	GetProcessTime
	GetLocalTime
HOST	gethostbyname
IP	gethostbyaddr
UID	getuid
VERSION	GetVersion
DIRECT	GetCurrentDirectory
	GetSystemDirectory
	GetWindowsDirectory
PID	GetCurrentProcessId
TID	GetCurrentThreadId
PROGRAM	getprogramname
ERRNO	GetLastError
ErrnoStr	strerror
Format	par_str
VAR	-

- **Event term:** A string constant describing the details of an event occurring in the system. In Fig. 1, `limiting connections by` is the event term.
- **Status term:** An integer variable often in a certain range denoting the status of the system. In Fig. 1, the number `304` represents a web connection status.
- **Variable term:** Fields in a log message representing string variable components such as file path, IP address, timestamp and so on. Variable terms are usually generated by calling external or internal subroutines in the binary code, and are useful for log-based analysis such as failure diagnosis.
- **Token:** A string constant showing as a prefix or suffix of a variable term indicating the content of the variable term. In Fig. 1, `host:` is a token.

Definition 2 (Log Template): A formatted abstraction of a log message using string constants and tags to indicate various components (terms) of the log message.

The log template that generated the log message in Fig. 1(a) is shown in Fig. 1(b), where the event term, status term, and token are reserved in the template, and variable terms are replaced with tags indicating their contents. The tags considered in this paper are listed in Table I, and can be extended and adjusted when applying to different binary codes. Hence, the aim of this paper is to extract the complete set of log templates from X86 binary codes.

B. Log Functions and Log Generation Forest

In order to extract log templates, we need to understand how a log message is produced. The logging process is usually initiated by a function call to a log generation function, which can appear in different modules and is triggered when a certain event happens. The logging process usually ends with an output function, such as a `printf` function or a third party library function such as output functions in `Log4j`. In general, we define all functions that are related to log generation as log functions.

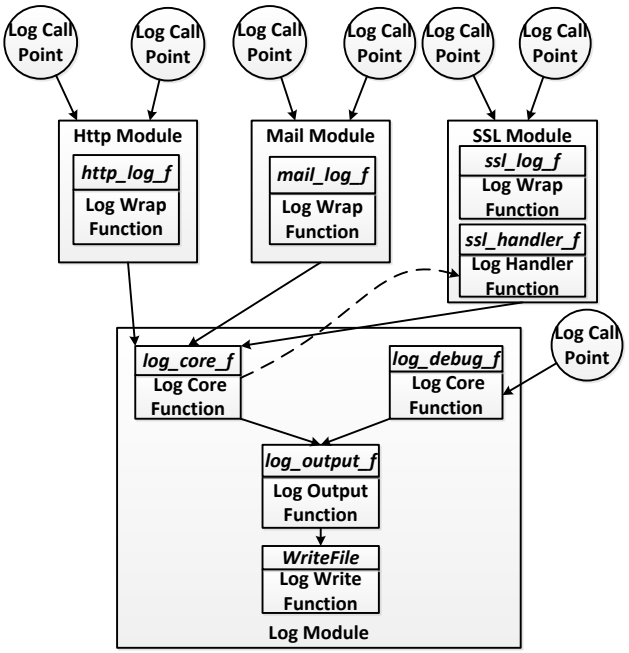

Fig. 2. Call Graph of Log Functions of a Web Server.

For some applications, the logging mechanism is simple, where features of a log are collected within a single function and outputted through a `printf` or `WriteFile` statement. For such case, identifying log template is relatively straightforward, and we just need to find all call points of the `printf` function, which can be identified in the import segment of the binary code. For a more general case, there is a separate log module in many applications, where functions are designed to be in charge of encapsulating log features, log output, and so on. As a result, the chain of function calls between the log initiation call and the final log output can be long, which makes template extraction difficult and challenge. We summarize the categories of log functions as follows:

- **Log output function:** A function that directly outputs a log message, such as `WriteFile`. We refer to it as f_{output}. A log output function can be a system call or a function from a third party, such as `Log4j`. If a log output function has wrapper functions, we also refer to the outermost wrapper function as f_{output}.
- **Log core function:** A function that combines all parts of a log message together, it prepares the various parts of a log message through system function calls and parameter passing, and connects different parts following a certain order, and finally calls a log output function to output the log message. We refer to it as f_{core}.
- **Log wrap function:** A function that does nothing but wraps a log core function, and may appear in various modules of the binary code. We refer to it as f_{wrap}.
- **Log handler function:** A function that is called by log core functions or wrap functions to compose some parts of a log message, or to connect various parts of a log message. We refer to it as $f_{handler}$.
- **Log generation function:** A function where the log generation begins. It can be a $f_{handler}$ which wrap a f_{core} or the f_{core} if it is called directly to generate logs. We refer to it as f_{gen}.
- **Log generation call point:** Log generation call points are where log generation functions are called, and serve as the entrance of the log generation function. At the call point, parameters are passed to log generation functions, from where parameters are passed and stitched by other log related functions, and ultimately different log messages are produced.

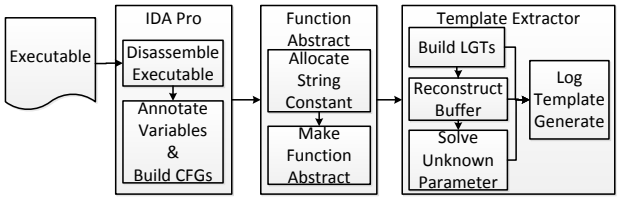

Fig. 3. Overview of GenLog.

We require a log template must be supported by a complete path including its call point, its core construction functions, and its output function. Note that for programs with simple logging strategy, f_{gen}, f_{core}, and f_{output} may be the same `printf` function, in which case a call point to `printf` defines a complete path for a log template. When complete paths of log templates merge at a common log function, we see a tree or a forest, which are defined as follows. A sample call graph of log functions of a web server is shown in Fig. 2.

- **Log generation tree (forest):** Log functions and their calling relationships define a log generation tree/forest, where log functions are vertices and their calling relationships are edges. Each log output function is the root node. Log core functions, wrap functions, handler functions and generation functions are internal nodes, and ench log generation call point is the leaf node. In a log generation tree, a path from the leaf node to the root node corresponds to the output of a log message. We refer to log generation tree and forest as T_{log} and F_{log}, respectively.

III. Methedology

A. Overview

The overview of our proposed log template extraction approach is shown in Fig. 3. We use IDAPro [14], [15], [31] to disassemble the target binary code to obtain its variables, call graphs, and control-flow graphs (CFGs). GenLog has two components, function abstract maker and template extractor. Since event terms and tokens of log templates are all string constants, they are essential for log template extraction and become an indicator for possible log functions. The function

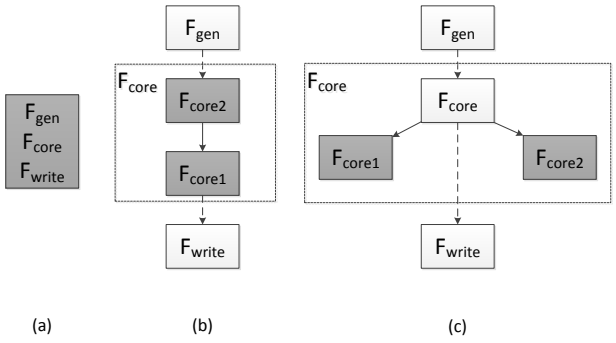

Fig. 4. Lattice of CC_{log}.

abstract maker of GenLog makes function abstracts only for the functions that have string parameters or have string constants in its body.

The main log template extractor has three steps:

- We first identify all log functions defined in section 3 using a combined bottom-up and top-down slicing method, and build a F_{log}.
- Second, we reconstruct buffers where log messages are prepared in log functions, using variable annotations provided by IDAPro and semantics of system calls.
- Third, we need to resolve unknown parameters using a token-aimed trimming algorithm to complete log template extraction. IDAPro cannot resolve all parameters passed to log functions due to unidentified indirect jumps and calls, complex data structures, and log handler functions.

B. Build Log Generation Forest

Due to the binary code variable type uncertainty, alias analysis and other complex situations, when we making the binary code analysis,there is a contradiction between accuracy and code size . Excessively accurate analysis methods are often not applicable to bulky binary codes. In this section, we use static slicing technique to separate the log correlation functions in the target binary code, which can effectively reduce the workload of code analysis, and can be applied to the larger binary code.

In order to retrieve all log correlation functions,a hybrid bottom-up and top-down slicing approach was proposed. We first identify all f_{output} and f_{core} candidates following certain rules, which can be considered as a bottom-up slicing step [9]. We then identify all f_{gen} candidates and explore their call chains, i.e., a top-down slicing step. Following the call chain, if we find a f_{gen} candidate reaches a f_{output} or f_{core} candidate, we confirm that these candidates are log functions and they form a branch in the $Tree_{log}$. In order to define the completeness of the hybrid slice approach for further description, we first introduce the log generation lattice.

1) Log generation lattice: A binary code can be expressed as $\mathcal{B} = \langle C, F\rangle, C = F \times F$, where F is the set of functions in the binary code and C denote the calling relationship between functions. $f_i \in F$ is a function, let $\overrightarrow{f_i f_j} \in C$ to denote that f_i call f_j.

The slicing algorithm is based on a completeness assumption: the log generation module is abstracted as a function call chain, denoted as $CC_{log} = \{f_i | f_i \in F\rangle$, taking the f_{gen} as the functional starting point, receiving event information, state information and variable information as parameters. f_{core} is the intermediate node, which is responsible for splicing the log messages. the final f_{output} output log messages. Based on the above assumptions, it is possible to define a bounded lattice for the CC_{log}.

Definition 3 (Log generation lattice): $(L_{log}, F, \vee, \wedge)$,$L_{log}$ is a lattice, the range of the lattice is F. The join operation $f_i \wedge f_j$ is the public predecessor function of f_i and f_j , denoted as f_{pre}, and the meet operation $f_i \vee f_j$ is the public successor function of f_i and f_j, denoted as f_{pre}. $\top$ and $\bot$ denote the top and bottom element.

Our work is using hybird slicing approach to get a set $S = \{f_i | f_i \in CC_{log}\}$, let $L_{log} \models S$.

Lemma 1: $L_{log} \models S$ is complete.

Proof:

1) $f_{gen} = \top, f_{output} = \bot, f_{gen}, f_{output} \in S$.
2) $\forall f_i, f_j \in S, f_i \wedge f_j = f_{pre}, f_i \vee f_j = f_{suc}, i, j \in \mathbb{N}$.
3) $if \quad \exists f_i, f_j \in S, f_i \wedge f_j = \{f_{pre}, f'_{pre}\}, \quad or \quad f_i \vee f_j = \{f_{suc}, f'_{suc}\}, \quad then \quad f_i, f_j, f_{pre}, f_{suc} \in S, \quad f_i, f_j, f'_{pre}, f'_{suc} \in S', S \neq S'$.

■

In the CC_{log}, any two functions f_i and f_j have the minimum upper bound and the maximum lower bound, respectively, their public predecessor function f_{pre} and public successor function f_{suc}. If they have more than one common function, f_{pre} or f_{suc}, without calling relationship between them, then they should belong to two different lattice.

The fitness of the completeness of L_{log} is:When we get a CC_{log} through the slicing algorithm, this CC_{log} must conform to the properties of the L_{log}. That is, the result of the slice must contain a f_{gen} as the minimum value for the lattice, and a f_{output} as the maximum value for the lattice. The f_{core} in the middle of the CC_{log} can consist of several functions, which can be partial ordering, but each paire of them must has the minimum upper bound and the maximum lower bound, or they will be sliced into two different CC_{log}. This can guarantee the f_{core} of the function of unity.Only to meet the above characteristics, a CC_{log} is fully functional, the call chain can be used as a branch of a $Tree_{log}$.

Fig. 4 shows several forms of L_{log}. The lattice can be a single function, as Fig. 4 a shows, or a function call chain. As the key to the log generation module, the form of the f_{core} may be more complex. It may has two functions, one calling another, as Fig. 4 b shows. Or several functions perform the main functions of the f_{core}, which have a common predecessor function on the CC_{log}, as Fig. 4 c shows. We can reduce this set of functions to a node on the CC_{log}, so that the CC_{log} still satisfies the definition of the L_{log}.

The following sections describe how to use the hybrid slice method to get call chains of log correlation functions that correspond to the L_{log}'s definition.

2) Bottom-up slicing: In this step, we first identify all f_{output} in the target binaries using a list of known system calls and library functions related to string output, (e.g., `fputs`, `fwrite`, `fprintf`, `write`), which can be prepared by searching through the import segment of the target binaries, we put them in a set named S_{output}. The f_{output} is the end point of the log generation call chain, but it is the most easily recognized function so it is the starting point for slicing. Sometimes, f_{output} will have a simple outer wrapper function, we will collectively referred to as f_{output}. Then in those functions which call the f_{output}, we determine the f_{core}.

We identify f_{core} condidates if they match the signature of a f_{core} function:

1) An f_{core} must have memory allocation operations.
2) An f_{core} must have string concatenate operations such as `sfprintf` or `vsfprintf`.
3) An f_{core} must have wildcard characters, such as `%`.
4) An f_{core} must call f_{output} to output log messages.

we add this function to the f_{core} candidate set S_{core} as a pending f_{core}. Then we trace back functions which call pending f_{core} to locate f_{wrap} and log call points. The call points of this pending f_{core} to find whether there are strings parameters passed in it. If so, then we can infer that it's successor function in S_{core} is a f_{core}. There may have some extra layers of function do nothing but wrapped the f_{core} and pass the string parameters to the f_{core}, we refer these functions as f_{wrap}, and the outermost layer of these f_{wrap} is f_{gen}. As we defined before, a f_{core} may be a f_{gen} if it has no f_{wrap}, or it may have more than one f_{gen} which is used at different call point of the binary. For example in Fig. 2, the function with shadow all can referred to as f_{gen}.

The search for all of the f_{gen} is difficult because that: during the process of tracing back, in each layer of the hierarchical, a function may be called by amounts of functions, if we go through with all of these invocation paths, the path explosion may occur. To avoid this complicated situation, we set a threshold as 10, and the tracing back process will stop when the number of functions in the call chain exceeds the given threshold. Very few programs having such many f_{wrap} will trigger this state, and then we go through step two to avoid this situation.

3) Top-down slicing: As is defined, a f_{gen} will pass string parameters to f_{core}. In this step, we retrieve functions at the top of the CFGs. We begin with functions which have been abstracted which has string parameters or has strings constants in their body. These functions referred to as pending f_{gen}, and is added to the f_{gen} candidate set S_{gen}.

We then use thses function as a starting point to retrieve down the call chain, if we can reach the functions in S_{core} that the bottom-up step had explored, then these functions can be comfrimed as f_{gen}, and the addresses at where these f_{gen} are called are log call points.

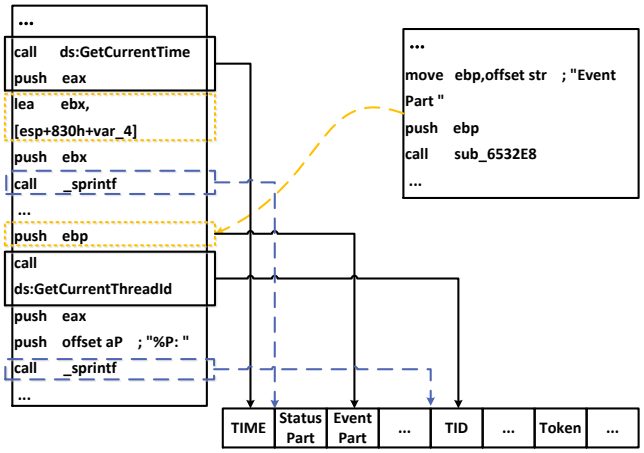

Fig. 5. Log Buffer Reconstruction. The lower rectangle is the log reconstructor buffer, on the left is code in f_{core}, on the right is code at log call point. Instruction in dashed rectangle denote the string operation, and codes in dotted rectangle denote unkown parameters.

In this step,a threshold also must be set to prevent the search path from exploding. If the length of all search paths starting from a pending f_{gen} exceeds the threshold, the pending f_{gen} will be removed from the S_{gen}. And if a pending f_{core} in S_{core} is not be reached in this step, it will be removed from the S_{core}.

Through the above slicing process, we will get many log generation call chains CC_{log}:

$$CC_{log} \mapsto CS = [f_1, f_2, ..., f_n],$$
$$f_1 \in S_{gen}, f_n \in S_{output}, i \in \mathbb{N}, L_{log} \models S.$$

Each call chain satisfies the completeness of the log generation lattice L_{log}, and can be as a branch of the $Tree_{log}$. As a result, joining the CC_{log} set CS together, we can get a $Tree_{log} = \bigcup CS_i, i \in \mathbb{N}$. The log call points is the leaf nodes, the f_{gen} and the f_{core} is the intermediate nodes, and the f_{output} is the root node.

C. Reconstruct Log Buffers

To extract an exactly log template, the order of all parts in the log template are necessary. As defined in section 3, f_{core} receive the structures of the log message and stitch them together. We assume that a log message is stitched in a log buffer which is constructured in f_{core}. We can extract log templates by rebuilding these buffers and retrieving string operations to record the log template parts order.

The memory buffer in f_{core} can be allocated once or times. We retrieve the pointer pointing to the buffer and record the position of each extension operation as well as string operation on the buffer, such like `call _sprintf`, as is shown in Fig. 5. These operations represent combination of log messages components. Then data flow analysis is been used to trace parameters passed in the f_{core}.

Log construction buffer need a space to load the strings as parts of log messages. So in a f_{core}, the return value

of the instructions, e.g. `call _malloc`, `sub epx offset`, which opening up a large stack space or calling memory allocation function would be the bengining address of a log construction buffer. And we denote $\texttt{Loc}_{\texttt{str}}$ as the address space of the log construction buffer, and $\texttt{Ref}_{\texttt{str}}$ as the reference to string type values. So a $\texttt{loc}_{\texttt{str}} \in \texttt{Loc}_{\texttt{str}}$ also can be a $\texttt{Ref}_{\texttt{str}}$.

A string operation instructions such as `call _sprintf` is the splicing node which stitch string type source parameters into a destination string, and then load it into the log construction buffer. As the string operation function are usually the lib function, so we can get the lib interface to know the stitch form of these source string parameters. And these destination strings will match the buffer operations one by one. According to this, the order of the parameters can be recorded. Those strings parameters can be referred to as event part of log template, the integer parameters can be referred to as status part.

So that we define a string operation on log construction buffer as:

$$\texttt{StrOp}\langle \texttt{loc}_{\texttt{buf}}, \texttt{args} \rangle, \quad \texttt{loc}_{\texttt{buf}} \in \texttt{Loc}_{\texttt{str}}, \texttt{args} \in \texttt{Ref}_{\texttt{str}}$$

$\texttt{loc}_{\texttt{buf}}$ is the start address of the str operation in the log construction buffer, args is a list of string reference $[\texttt{arg}_1, ..., \texttt{arg}_\texttt{n}]$, which are the parameters of the StrOp, n is the number of parameters. According to the lib interface, we could know which arg has wildcard characters `"%"`, which will be replaced by other args.

Going through the control flow of the f_{core}, we can get the partial order relation of these StrOp, then we can get a ordered list $[\texttt{StrOp}_1.\texttt{loc}_{\texttt{buf}} < ... < \texttt{StrOp}_\texttt{n}.\texttt{loc}_{\texttt{buf}}]$ based on StrOp's partial order, then a log construction buffer can be defined as:

$$[\texttt{args}_{\texttt{buf}}1 = \texttt{StrOp}_1.\texttt{args}, ..., \texttt{args}_{\texttt{buf}}\texttt{n} = \texttt{StrOp}_\texttt{n}.\texttt{args}],$$
$$\texttt{StrOp}_1.\texttt{loc}_{\texttt{buf}} < ... < \texttt{StrOp}_\texttt{n}.\texttt{loc}_{\texttt{buf}}$$

if loops exist, we set a boundary of the loop as 10.

We can get log construction buffers in binary code by retriving all f_{core} discovered, then we will fill the buffer with the values of args.

Now that $\texttt{Buffer}[\texttt{args}_{\texttt{buf}}1, ..., \texttt{args}_{\texttt{buf}}\texttt{n}]$ based on string operation is constructed, we need to parse the args of each part in the log buffer to get the log template. We parse the parameters into two steps using data flow analysis:

1) as we have sliced the CC_{log} each of which is a complete log call environment, we retrieve data flow from the log call point ,through CC_{log}, down to f_{core}. The we can get a map that from parameters of f_{gen} to $\texttt{args}_{\texttt{buf}}$.
2) we then solve the parameters of f_{gen}, if the value of a parameter is ambiguous, we will solve it out of the CC_{log} to determine the value.

In the fist step, we identify type and number of parameters of each f_{gen} in advance. We use $p_i[tag_1, ..., tag_n]$ to denote the prototype of function f_i, n is the number of parameters of f_i. In x86 binaries, parameters are passed by the instruction `push` throngh the stack, according to the calling convention, the number of `push` before the `call` insttruction denote the number of parameters. Some types of parameters are well identified. If the operand of the `"push"` instruction is a integer, we label the parameter with tag `"INT"`. IDAPro also can identify the reference of a string constant, so if the operand is $\texttt{Ref}_{\texttt{str}}$, we label the parameters with tag `"STR"`, and label the parameters with tag `"VAR"` for other parameters which can not be identified so far.

As a f_{gen} will be called at many place, we search every call point of a f_{gen}, if a string type of value is assaigned to a parameter at one place, we can determine that parameter's type. so that f_{gen}'s prototype can be identified. We can idnetify the prototype of f_{core} as well.

We then define a parse funtion f_{parse} to parse parameters of one function passing through a call chain to variables in another function as :

$$\begin{aligned} f_{parse}\langle f_i, f_j \rangle \mapsto & f_i(arg_1, ..., arg_m) \\ & f_j(arg_1, ..., arg_n) \\ & fvar_j(var_1, ..., var_n) \\ & arg_j k = arg_i k, \\ & var_j k = arg_j k, \\ & 1 < k < n, m \end{aligned}$$

so we can use a $\texttt{f}_{\texttt{parse}}\langle \texttt{f}_{\texttt{gen}}, \texttt{f}_{\texttt{core}} \rangle$ to get a $\texttt{map}_{\texttt{gen}}\langle \texttt{arg}_{\texttt{gen}}, \texttt{var}_{\texttt{core}} \rangle$ from parameters of $\texttt{f}_{\texttt{gen}}$ to the variable in $\texttt{f}_{\texttt{core}}$. Then we can get a $\texttt{map}_{\texttt{core}}\langle \texttt{var}_{\texttt{core}}, \texttt{args}_{\texttt{buf}} \rangle$ using data fllow analysis in $\texttt{f}_{\texttt{core}}$, the joining $\texttt{map}_{\texttt{gen}}$ and $\texttt{map}_{\texttt{core}}$ using $\texttt{map}_{\texttt{gen}} \vee \texttt{map}_{\texttt{core}}$, we can get $\texttt{map}_{\texttt{buf}}\langle \texttt{args}_{\texttt{gen}}, \texttt{args}_{\texttt{buf}} \rangle$. So that the value of the log construction buffer passed in the corresponding $\texttt{CC}_{\texttt{log}}$ can be determined. As the slice of the $\texttt{CC}_{\texttt{log}}$ is very small, the parse process will take little time and will not face complex situations.

D. Solve Unknown Parameters

As discript in subsection C, variables in $\texttt{args}_{\texttt{buf}}$ can be labled with tags such as `"INT"`, `"STR"`, and `"VAR"` by the map $\langle \texttt{arg}_{\texttt{gen}}, \texttt{args}_{\texttt{buf}} \rangle$. If a variable which reference a string dirctly or is a interger, we can get its value directly. But some variable labled with `"STR"`, but it's fixed value point is a instruction like `push ebx`, so that it is difficult to determin this variable's value, but to get accurate log templates, we must solve these paramters efficient.

In a striped binary, because of lack of type information and alias problems, solving unknown parameters left in log functions is difficult [3], [5], [16], [23]. Furthermore, these unknown parameters usually are either passed by registers, or by indirect addresses that IDAPro cannot recognize, which often point to complex data structures or log handler functions, as is show in Fig . 5.

If an unknown parameter labled by `"STR"` or `"INT"` is passed by a register, e.g., `pushebx`, we need to find the latest direct address assignment recursively. For example, a sample source code and its assembly code from IDAPro are shown in Figure 6. At line 106 of the assembly code, we cannot infer the value of `eax` in a case of an instruction `moveeax`, until an instruction `call eax` or `move eax, STR` is discovered.

Fig. 6. Example of Parameter Solving.

It can be more complicated to solve an unknown parameter which is a complex type of aggregation, which the elements may be the tokens of log templates or the address of $f_{handler}$ being assigned far away.

The access on a structure is always performed through an address expression of the form [`base` + `offset`], where `base` is a register and `offset` is an integer constant. And `base` represented the address of the structure and each `offset` represented the address of an element of the structure. Some research projects have given some more precisely algorithms [5], [30] to analysis the elements of the structure, e.g., in Fig. 6, all alias set of `ebx`, `ebx` + `offset`, `ecx`, `ecx` + `offset` must be analysis. In large binary code, such an analysis will cause the alias analysis to explode.

We use a token-aimed trimming to solve this problem. In our case, we just focus on elements assigned with string constants which represent tokens. Because of that the assignment of the structures elements might be very far away, if we use a precise reaching definitions analysis, we might go throng most of the binary. As token is the most interesting thing which is assigned to a structure or in a log handler function, we can only focus on whether a given variable can reach an assignments where a token is assigned to it, and ignore the influence bringing from assignments by alias string-independent. For example, at Line 96 in Fig. 6, we just focus on whether the ebx can be assigned a value of tokens (at line 12), and ignore the possibility that after the assignment (at line 28), although the value of ebx might be changed by its alias.

The retrieval of the $f_{handler}$ is in the same case. A function will be referred to as a $f_{handler}$ by checking that whether it has tokens operation. As a $f_{handler}$ might be defined far away from the CC_{log}, as shown in Fig. 6, we are interesting in whether the given register in a call instruct, (e.g. `calleax`), can get to a instruction in which the address of a $f_{handler}$ is assigned to it. And we do not analysis the pointers alias which is created along the tracing paths.

We also set a threshold as 5 to limit the search scale, a search will trace back followed the data flow not more than 10 times of functions crossing to avoid path explosion. If a variable can not be sloved, we lable it as "`VAR`", and if it can be sloved, we lable it as "`STR`" or "`INT`" accordingly.

Furthermore, the unkown variable can be a return value of a lib function with special semantic, which usually is the variable part of log messages. The semantic of these fields can bring benefit for log analysis, and help us to slove the variable part of log messages.

The intuition behind our variable part inference is that many functions used by programs contain rich semantic information. We can leverage this information to infer field semantics including timestamps, filenames, hostnames, ports, IP addresses, and many others. The semantic information of those functions is publicly available in their prototypes, which describe their goal as well as the semantics of its inputs and outputs. We can get detail information refer to system manufacturers manuals accordingly.

In our approach, a semantics function table had been established in advance (Table I). Taint propagation is been used to find whether the unkown variable is the return value of call a semantics function in Table I. If so, we use the functions prototype to check if any of the arguments in the buffer is tainted. These tainted arguments will match the fields of log templates variable parts. Then we will labeled the variable with a semantic tag accordingly.

After all the process above has been done, we will get parts of log templates with string constants, semantic tags. We use regular expression to descript log templates, such as "`[PROGRAM]:[Format][VAR][*ErrnoStr]`". For some lib functions generating log message, these integer parameters will be replaced into strings, one number match a statment, we use tag `ErrnoStr` to denote these statment. And if we cannot slove the number of the integer parameter, we use "`[*ErrnoStr]`" to denote that each statment in the match table is possible.

IV. Experiments

We evaluated GenLog on four applications (including one server), all of which were complied in C. They cover a wide range of applications including GNU coreutils, command-line network connection tools, and web servers. Table II summarizes these applications.

TABLE II
Applications evaluated in our experiments.

Application	Version	Format	Dsecription	Size
mkdir	8.13	ELF	GNU coreutils	42KB
tar	1.28	ELF	GNU coreutils	299KB
plink	0.67	PE	open source	4,582KB
Nginx	1.6.0	PE	web server	30,234KB

We conducted experiments on a 64-bit Win7 computer with Intel 3.40GHz i7-3770 CPU and 8GB RAM. GenLog was implemented using Python 2.7, IDA Pro 6.1, and idapython 1.5.2.

A. Log Template Evaluation

For mkdir, tar, and plink, we obtained the log file by executing the applications. For Nginx, we obtained the log file from [1]. Table III shows the size of the log file and the number of logs for each application. Furthermore, we

manually extracted log templates from each log file and the total number of log templates are shown in Table III as well.

TABLE III
LOGS EVALUATED IN OUR EXPERIMENTS.

Application	Log Size	# of Logs	# of LTs	Source
mkdir	13KB	306	7	program execution
tar	15KB	397	15	program execution
plink	28KB	702	20	program execution
Nginx	96,182KB	363485	53	[1]

To evaluate the effectiveness of GenLog, we used $Precision_L$ and $Recall_L$ to measure the precision and recall of the log templates extracted by GenLog for logs in each log file, which are defined as follows:

$$Precision_L = \frac{L_m}{L_m + L_f}, \tag{1}$$

$$Recall_L = \frac{L_m}{L}, \tag{2}$$

where, L is the total number of logs in a log file, L_m represents the number of logs whose manually extraced log templates are matched with the ones found by GenLog. We say two log templates match when they have the same event, status, variable parts, and tokens, and in the same order. L_f represents the number of logs for whom GenLog discovered the event part in their log templates but not with a complete match. Furthermore, we use L_{nf} to represent the number of logs for whom GenLog failed to discover their log templates.

We also used $Precision_T$ and $Recall_T$ to measure the precision and recall of the log templates extracted by GenLog for unique log templates in each log file, which are defined as follows:

$$Precision_T = \frac{T_m}{T_m + T_f}, \tag{3}$$

$$Recall_T = \frac{T_m}{T}, \tag{4}$$

where, T is the total number of unique log templates in a log file, T_m represents the number of manually extraced log templates are matched with the ones found by GenLog. T_f represents the number of log templates for whom GenLog discovered the event part but not with a complete match. Furthermore, we use T_{nf} to represent the number of log templates that GenLog failed to discover.

B. Overtall Results

We first show the overall performance of GenLog on four datasets in Table IV. We can see that GenLog successfully discovered all log templates present in the log file of mkdir, tar, and plink. For Nginx, GenLog found 46 out of 53 log templates with a perfect match, found the event part of 6 log templates, and missed 1 log template. The $Precision_L$ and $Recall_L$ values are higher than $Precision_T$ and $Recall_T$, because common logs tend to have simple templates and can be found by GenLog easily.

TABLE IV
OVERALL RESULT.

Logs					
Application	L_m	L_f	L_{nf}	$Precision_L$	$Recall_L$
mkdir	302	0	0	100%	100%
tar	397	0	0	100%	100%
plink	702	0	0	100%	100%
Nginx	363428	52	5	99.9%	99.9%

Log Templates					
Application	T_m	T_f	T_{nf}	$Precision_T$	$Recall_T$
mkdir	7	0	0	100%	100%
tar	15	0	0	100%	100%
plink	20	0	0	100%	100%
Nginx	46	6	1	88.4%	86.7%

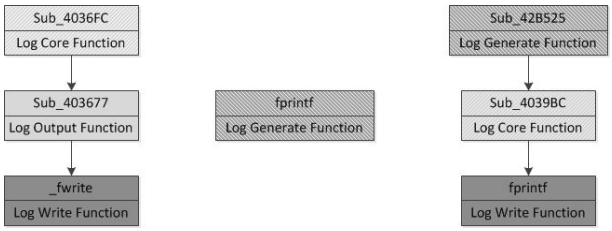

Fig. 7. Log Generation Forest of plink

C. Case Studies

mkdir mkdir is an application in GNU coreutils to create file directories. GenLog found mkdir produces logs using `_error` from glibc library to generate logs, and there are 17 call points in total. The log templates present in the log file of mkdir are all correctly found by GenLog, which are shown in Table VII.

TABLE V
EXTRACTED LOG TEMPLATES FOR MKDIR

No.	Log Template	Matched
1	`"mkdir: cannot create directory [VAR]: File exists "`	True
2	`"mkdir: invalid option [VAR]"`	True
3	`"mkdir: missing operand [VAR]"`	True
4	`"mkdir: write error [VAR] [VAR]"`	True
5	`"mkdir: memory exhausted"`	True
6	`"mkdir: failed to set default file creation context to [VAR]"`	True
7	`"mkdir: cannot change permissions of [VAR]"`	True
8	`"mkdir: cannot change owner and permissions of [VAR]"`	True

plink plink is a command-line network connection tool. Figure 7 shows the log generation forest found by GenLog

TABLE VI
PROTOTYPES OF LOG GENERATION FUNCTIONS OF PLINK

Function	Prototype	# of Calls
`sub_4036FC`	(`void` $*$ `Var`, `constchar` $*$ `Format`, `args`)	11
`sub_42B525`	(`void` $*$ `Var`, `constchar` $*$ `Format`, `args`)	152
`fprintf`	(`constchar` $*$ `Format`, `args`)	23

GenLog found three log generation trees in plink as shown in Figure 7. Table VI lists three log generation functions,

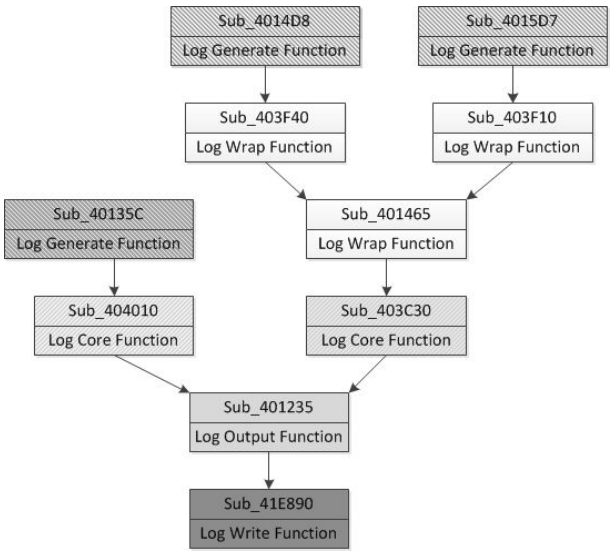

Fig. 8. Log Generation Forest of Nginx

their prototypes, and the number of call points, among which `fprintf` is both a log core function and a log write function.

The log templates present in the log file of plink are all correctly found by GenLog. Table VII shows some of the sampled extracted Log Templates by GenLog.

TABLE VII
SAMPLED EXTRACTED LOG TEMPLATES FOR PLINK

No.	Log Template	Matched
1	`"Host not found."`	True
2	`"The handle passed to the function is invalid."`	True
3	`"The target was not recognized."`	True
4	`"Network error: Connection timed out."`	True
5	`"Network error: Network dropped connection on reset."`	True
6	`"Network error: Address family not supported by protocol family."`	True

Nginx Nginx is a web server and has a log module and complicated data structures to handle logs. Figure 8 shows the log generation forest found by GenLog, which only contains one tree that corresponds to its log module. In Figure 8, we show various log write functions, log core functions, and log generation functions. We also show the three log generation functions, their prototypes, and the number of call points in Table IX.

For Nginx, GenLog found 46 out of 53 log templates with a perfect match, found the event part of 6 log templates, and missed 1 log template. Table VIII shows some of the sampled log templates extracted by GenLog. We can see that GenLog is able to extract complicated log templates, with various event terms, such as `"client:"`, `"server:"`, `"request:"`, `"host:"`, and variable terms, such as `[TIME]`, `[PID]`, `[TID]`, and `[DIRECT]`. The last two rows are two log templates that GenLog failed to discover completely. The mismatched part is shown in bold face. For both cases, GenLog cannot resolve these string constants due to the limit on the depth of resolving unknown parameters.

D. Performance of GenLog

Finally, we show the running time (in second) of GenLog for various applications in Table X, which also includes the size of each binary code and the number of functions found by IDAPro. We can see that for small binaries GenLog can finish under 2 minutes, and almost scale linearly to larger size of binaries.

V. RELATED WORKS

Log template discovery is the prelude for failure diagnose of applications. Log message mining projects have existed for a long time. Those projects often relied on frequent pattern mining and cluster analysis [11], [19], [25], [33] which all depend on the quality of the log files. They can only get the template appeared in log files and could hardly find out the log rarely appeared.

Yuan Ding et al [38] and Nguyen Hiep et al [27] match the log template extracting from source codes with log files to narrow down the set of the execute paths. But source codes are not often available, and we must get the right version of the source code matched the executables generating the log files. Our approach is based on stripped binaries without using log files and source codes. Executables are always at hand and easer to gain.

Our work is also related to binary reverse engineering [4], [6], [8], [21], [24], [29], [39] and automatic protocol reverse techniques [7], [9], [13]. Lim et al [10] use inter-procedural static analysis to extract the format from application data which output by a program. To use their method, human intervention is needed to identify output functions and to assign higher-level interpretations to selected fields identified by the analysis. Our approach differs in that we do not require any knowledge about the program before analysis.

Mihai Christodorescu and Nicholas Kidd [12] proposed a string-analysis tehnique that recovers C-style strings in x86 executables. Their method requires the user to provide points of interest in an executable program as the beginning of the analysis. And they build their analysis on an existing work called Java String Analyzer(JSA). Our approach do not need a given start point and we build the log construction buffer to perform the string analysis.

Caballero et al [33] proposed an automatic technique for extracting encryption routines from binary. They use a dynamic approach to identify the component of web protocol. But dynamic analysis can only get information from the instruments in execution paths which depend on a special input. Our approach use static analysis that can go through all parts of binaries without designing an input.

Slice technology is a commonly used code analysis technology [22], [32], [35], [36]. In our approach, we use a top-down and bottom-up slcing method to get all log-related functions in binary code, and this is like the work Xin Zhang, Ravi Mangal, Mayur Naik and Hongseok Yang [40] performed. By using

TABLE VIII
SAMPLED EXTRACTED LOG TEMPLATES FOR NGINX

No.	Log Template	Matched
1	"[TIME] [VAR] [PID]#[TID]: could not respawn worker."	True
2	"[TIME] [VAR] [PID]#[TID]:[VAR] CreateDirectory() [DIRECT]"	True
3	"[TIME] [VAR] [PID]#[TID]: [VAR] access forbidden by rule, client: [VAR], server: [VAR], request: [VAR], host: [VAR]."	True
4	"[TIME] [VAR] [PID]#[TID]: upstream timed out (110: Connection timed out)while reading upstream, client: [VAR], server: [VAR], request: [VAR], host: [VAR]."	True
5	"[TIME] [VAR] [PID]#[TID]: [VAR] limiting connections by zone [VAR], client: [VAR], server: [VAR], request: [VAR], host: [VAR]."	True
6	"[TIME] [VAR] [PID]#[TID]: [VAR] open() [VAR] failed (2: No such file or directory), client: [VAR], server: [VAR], request: [VAR], host: [VAR]."	True
7	"[TIME] [VAR] [PID]#[TID]: [VAR] upstream prematurely closed connection while **reading response header from pstream**, client: [VAR], server: [VAR], request: [VAR], upstream: [VAR], host: [VAR]"	False
8	"[TIME] [VAR] [PID]#[TID]: [VAR] send() incomplete **while resolving [VAR], resolver: [VAR]**"	False

TABLE IX
PROTOTYPES OF LOG GENERATION FUNCTIONS OF NGINX

Function	Prototype	# of Calls
Sub_4014D8	(intState, void * Var, intState, constchar * Format, args)	833
Sub_4015D7	(intState, void * Var, intState, constchar * Format, args)	658
Sub_40135C	(intState, constchar * Format, args)	12

TABLE X
PERFORMANCE OF GENLOG

Application	Size	# of Functions	Time
mkdir	42KB	192	12s
tar	299KB	958	85s
plink	4,582KB	1057	123s
nginx	30,234KB	6173	976s

hybrid chips, the scalability of program analysis is enhanced, we can analyse large binary code. The difference is we use a lattice to ensuring the completeness of log function slices.

VI. CONCLUSION

We proposed a new approach to extract log template from stripped C-style binaries. Our approach do not need source codes and any debugger information in the binaries. We use a hybrid slice step building log generation tree to reduce the scope of code analysis, and reconstruct log buffer and then slove unkown parts of the buffer to get precise log templates.

ACKNOWLEDGMENT

This work is funded by the National 863 Program of China (Grant No. 2015AA01A301).

REFERENCES

[1] http://opensource.indeedeng.io/imhotep/docs/sample-data/.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499, 1994.

[3] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86a platform for analyzing x86 executables. In *International Conference on Compiler Construction*, pages 250–254. Springer, 2005.

[4] G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. *University of Wisconsin*, 2005.

[5] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 5–23, 2004.

[6] G. Balakrishnan and T. W. Reps. DIVINE: discovering variables IN executables. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 1–28, 2007.

[7] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, N. Tawbi, et al. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001(184-189):79, 2001.

[8] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 267–277, 2011.

[9] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010.

[10] J. Caballero, P. Poosankam, C. Kreibich, and D. X. Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 621–634, 2009.

[11] P. Chen, Y. Qi, P. Zheng, and D. Hou. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 1887–1895, 2014.

[12] M. Christodorescu, N. Kidd, and W. Goh. String analysis for x86 binaries. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*, pages 88–95, 2005.

[13] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *1997 International Conference on Software Maintenance (ICSM '97), 1-3 October 1997, Bari, Italy, Proceedings*, page 188, 1997.

[14] I. P. Disassembler. Debugger, 2010.

[15] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[16] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 51–60, 2013.

[17] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, pages 149–158, 2009.

[18] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, pages 24–33, 2014.

[19] S. Hommes, T. Engel, et al. Classification of log files with limited labeled data. In *Proceedings of Principles, Systems and Applications on IP Telecommunications*, pages 1–6. ACM, 2013.

[20] L. Huang, X. Ke, K. Wong, and S. Mankovski. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 conference of the Centre for Advanced Studies on Collaborative Research, November 1-4, 2010, Toronto, Ontario, Canada*, pages 313–326, 2010.

[21] E. R. Jacobson, N. E. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11, Szeged, Hungary, September 5-9, 2011*, pages 1–8, 2011.

[22] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *2011 IEEE Symposium on Security and Privacy*, pages 347–362. IEEE, 2011.

[23] U. P. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009.

[24] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1128–1139, 2014.

[25] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009*, pages 1255–1264, 2009.

[26] K. Nagaraj, C. E. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 353–366, 2012.

[27] H. Nguyen, D. J. Dean, K. Kc, and X. Gu. Insight: In-situ online service failure path inference in production computing infrastructures. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 269–280, 2014.

[28] D. Ogle, H. Kreger, A. Salahshour, J. Cornpropst, E. Labadie, M. Chessell, B. Horn, J. Gerken, J. Schoech, and M. Wamboldt. Canonical situation data format: The common base event v1. 0.1. *IBM Corporation*, 2004.

[29] J. Qiu, X. Su, and P. Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Trans. Software Eng.*, 42(2):187–202, 2016.

[30] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, pages 16–35, 2008.

[31] D. Rescue. Ida pro disassembler, 2006.

[32] M. Sridharan, S. J. Fink, and R. Bodík. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 112–122, 2007.

[33] R. Vaarandi. Mining event logs with slct and loghound. In *NOMS 2008-2008 IEEE Network Operations and Management Symposium*, pages 1071–1074. IEEE, 2008.

[34] R. Vaarandi et al. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, pages 119–126, 2003.

[35] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 253–264, 2010.

[36] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[37] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 117–132, 2009.

[38] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 143–154, 2010.

[39] J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 1*, pages 501–508, 2007.

[40] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *ACM SIGPLAN Notices*, volume 49, pages 249–258. ACM, 2014.