# GenLog: Accurate Log Template Discovery for Stripped X86 Binaries

***Abstract*—Computer systems often fail due to many reasons such as software bugs or administrator errors. Log analysis is an important method of computer failure diagnosis. With the ever increasing size and complexity of logs, the task of analyzing logs has become cumbersome to carry out manually. For this reason, recent research has focused on automatic analysis techniques for large log files. However, log messages are texts with certain formats and it is very challenging for automatic analysis to understand the semantic meanings of log messages. The current state-of-the-art approaches depend on the quality of observed log messages or source code producing these log messages. In this paper, we propose a method *GenLog* that can extract log templates from stripped executables (neither source code nor debugging information need to be available). GenLog finds all log related functions in a binary through a combined bottom-up and top-down slicing method, reconstructs the memory buffers where log messages were constructed, and identifies components of log messages using data flow analysis and taint propagation analysis. GenLog can be used to analyze large binary code, and is suitable for commercial off-the-shelf (COTS) software or dynamic libraries. We evaluated GenLog on four X86 executables and one of them is Nginx. The experiments show that GenLog can identify the template for log messages in testing log files with a precision of $99.9\%$.**

## 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 an 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 computing infrastructure and software, 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 drawn considerable attentions [2], [16]–[20], [25], [26], [34], [37], [38].

There are two challenges 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 requirement to be supported by at least 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 a method, meanwhile, it may be 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 (COTF) software, we could not obtain the source code. Even for open source software, different versions may lead to a change of log generation infrastructure, requiring an exact version of the software to be analyzed. 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 consuming and not applicable to large applications.

In this 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 are normally easier to obtain than source code. $(ii)$ GenLog is able to produce call points in the program for each log template, which can be used to reduce the search space of path inference tools, such as SherLog. GenLog employs an overall analysis of binary code 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, as well as alias confusions.

To summarize, we make the following contributions.

1) We define the characteristics of log related functions and define a complete path that ensures a log template including 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.

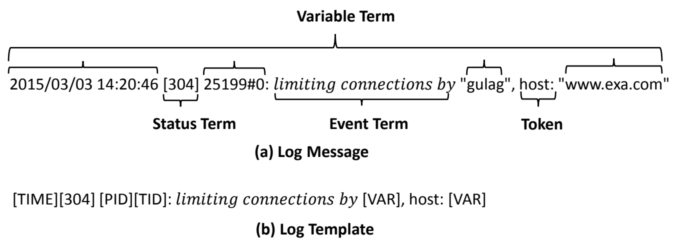

Variable Term
2015/03/03 14:20:46 [304] 25199#0: limiting connections by "gulag", host: "www.exa.com"
Status Term
Event Term
Token
(a) Log Message
[TIME][304] [PID][TID]: limiting connections by [VAR], host: [VAR]
(b) Log Template

Fig. 1: Log Message and Log Template.

2) GenLog reconstructs the memory buffer in which log messages were constructed, identifies components of log messages using data flow analysis and taint propagation analysis, analyzes alias using tag-aimed trimming to form log templates.
3) We report experimental results of GenLog on four X86 executables, one of them is Nginx. The experiments show that GenLog can identify the template for log messages in the testing log file with a precision of 99.9%.

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 Definition

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

- **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 a file path, an IP address, a timestamp, and so on. Variable terms are usually generated by calling external or internal subroutines in the binary code and useful for log-based analysis such as failure diagnosis.
- **Token:** A string constant appearing as a prefix or suffix of a variable term indicating the content of the variable term or connecting the variable term. In Fig. 1, "`host:`" is a token.

TABLE I: Tags of Variable Term

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

*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 tokens are reserved in the template, and variable terms are replaced with tags that indicate their contents. The tags considered in this paper are listed in Table I, and can be extended and adjusted when applying to different binary code. Hence, the aim of this paper is to extract the complete set of log templates from stripped X86 binaries.

### B. Log Functions and Log Complete Path

In order to extract log templates, we need to understand how a log message is produced. The log generation process is usually initiated by a function call to a log generation function, which can appear in different modules and be 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 refer to all functions that are related to log generation as *log functions*. For example, sample log functions of program `shttp` are shown in Fig. 2, where line 3 of function `ServiceMain` calls a log generation function `_shttp_elog`.

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 a case, identifying log template is relatively straightforward, and we only 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. For example, in Fig. 2, function `_shttp_elog` encapsulates log features and is called in function `ServiceMain`. Note that, the

```
1   void ServiceMain(long client_addr)
2   {
                                                ⋮
3        _shttpd_elog(file, client_addr, "Cannot initialize SHTTP context");
                                                ⋮
4   }

5   void _shttp_elog(File *file, long client_addr, const char *fmt)
6   {
7         char buf[1024];
8         int len;
9         ip = inet_ntoa(client_addr);
10        len = snprintf(buf, sizeof(buf), "[error][client %s]", ip);
11        sprintf(buf + len, sizeof(buf) – len, fmt);
12        shttp_write_fd(file, buf)
13  }

14  void shttp_write_fd(ngx_fd_t fd, void *buf)
15  {
16       (void)write(fp, buf);
17  }
```

Fig. 2: Sample Log Functions of Program `shttp`.

chain of function calls between the log initiation call and the final log output can be long, which makes template extraction difficult and challenging.

GenLog indentifies these log functions from binaries, which are characterized as follows:

- **Log output function:** A function that directly outputs a log message, such as `WriteFile`. We refer to it as an $f_{output}$. A log output function can be a system call or a function of a third party, such as `Log4j`. If a log output function has wrapper functions, we also refer to the outermost wrapper function as an $f_{output}$.
- **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, connects different parts following a certain order, and finally calls a log output function to output the log message. We refer to it as an $f_{core}$.
- **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 an $f_{wrap}$.
- **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 an $f_{handler}$.
- **Log generation function:** A function where the log generation begins. It calls an $f_{wrap}$ or $f_{core}$ function and passes parameters. It can also be an $f_{wrap}$ or an $f_{core}$ itself. We refer to it as an $f_{gen}$.
- **Log generation call point:** A call points where an $f_{gen}$ is called. It is 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. We refer to it as a $cp$.

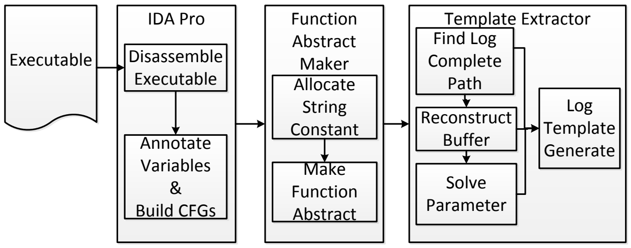


Fig. 3: Overview of GenLog.

We require that a log template must be supported by a complete path, which contains a log genration call point, a log generation function, a log core function, and a log output function. We define log complete paths formally as follows:

- **Log complete path:** A log complete path, denoted as $p$, is a call chain defined on a log generation call point and log functions, where log functions are nodes and their calling relationships are edges. It starts with a log generation call point, passes through log generation functions, wrap functions, handler functions, and core functions as intermediate nodes (wrap and handler functions are optional), and ends with a log output function.

For example, in Fig. 2, function `shttp_write_fd` is an $f_{output}$, function `_shttp_elog` is both an $f_{gen}$ and $f_{core}$. Line 3 in function `ServiceMain` is its call point. The log complete path is from line 3 of `ServiceMain` to `_shttp_elog` and ends with `shttp_write_fd`, which prepares and outputs log messages in the format of `"[error][client IP] Cannot initialize SHTTP context"`. Note that for programs with simple logging strategy, a single `printf` function may serve as an $f_{gen}$, an $f_{core}$, and an $f_{output}$ at the same time, in which case a call point to `printf` defines a complete path for a log template.

## III. Methodology

### A. Overview

The overall architecture of our proposed log template extraction approach is shown in Fig. 3. We use IDA Pro [14], [15], [31] to disassemble the target binary code to obtain its variables, call graphs, and control-flow graphs (CFGs). GenLog has two components, the function abstract maker and the template extractor.

Based on the fact that all event terms and tokens of log templates are in the format of string constants, they are essential for log template extraction. Thus, the function abstract maker of Genlog generates function abstracts on string constants, string parameters, and their references inside a function.

As for the main log template extractor, there are three steps to go through:

- `Step1` : GenLog identifies all log functions defined in section 3 using a combined bottom-up and top-down slicing method, and find all log complete paths $\mathcal{P}$.

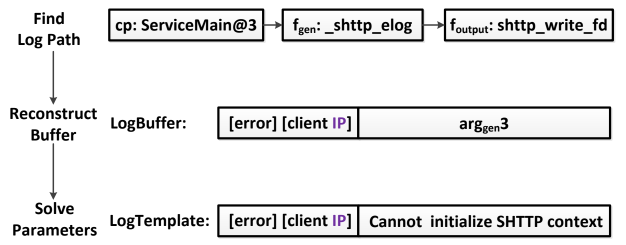


Fig. 4: GenLog Steps for the Sample Code in Fig. 2.

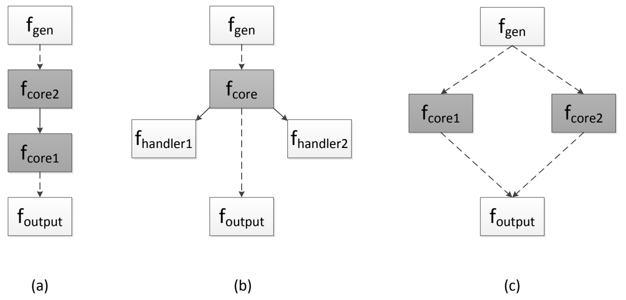


Fig. 5: Examples of Log Complete Paths.

- Step2 : GenLog reconstructs all log message buffers $\mathcal{B}$ based on $\mathcal{P}$ by following string operations and identifying components of log messages using data flow analysis, taint propagation analysis, and semantics of lib functions.
- Step3 : GenLog solves the parameters passed at each call point $cp$ to fill in its corresponding log buffers, to generate all log templates $\mathcal{T}$.

For example, Fig. 4 shows the log complete path, the log buffer, and the log template generated at each step by GenLog for the sample code in Fig. 2.

### *B. Find Log Functions and Log Complete Paths*

Extracting log complete paths from the target binary code has two purposes: 1) a log complete path ensures possible log templates and it is the base for log template construction; 2) separating log related functions from the target binary code can effectively reduce the workload of further code analysis, making it applicable to binary code of larger size.

The target binary code is represented as an annotated call graph $G = (V, E)$, where each vertex $f$ in $V$ corresponds to a function, and each edge $e =< f_1, f_2 >$ in $E$ represents function $f_1$ calls $f_2$. For each vertex $f$ in $V$, $f.code$ stores its assembly code line by line returned by IDA Pro, $f.abstr$ stores its function abstract generated by GenLog, $f.tag$ holds a tag indicating its type as one or more of $f_{gen}, f_{handler}, f_{wrap}, f_{core}$, and $f_{output}$. For our convenience, we also use a reversed graph $G' = (V', E')$ of $G$, where $V' = V$, and for each $< f_1, f_2 >\in E$ we have $< f_2, f_1 >\in E'$, which means $f_2$ is called by $f_1$.

Given a target binary code $G$ and its $G'$, we propose a hybrid bottom-up and top-down slicing approach to extract all log complete paths in $G$.

*1) Bottom-up slicing:* Although $f_{output}$ functions are the end of log complete paths, they are the easiest to recognize, because they are mostly system calls and known library functions related to string outputs. As a result, we prepare a list $L_{output}$ of possible output system calls and library functions, (e.g., `fputs`, `fwrite`, `fprintf`, `write`), and implement a function called `FindOutput`(), which identifies all $f_{output}$ in the target binaries by searching each vertex $f$ in $G$ to match $f.code$ with $L_{output}$, and the matched functions are labeled with $f_{output}$ and put in a set named $S_{output}$. In some cases, an $f_{output}$ has a simple outer wrapper function, to which we collectively referred as an $f_{output}$ as well.

Then, the bottom-up slicing algorithm finds $f_{core}$ functions by initiating a breadth-first search (BFS) from each $f_{output}$ in $S_{output}$ on $G'$ and checking the following criterion for a $f_{core}$ function:

1) It or its non $f_{core}$ successors in $G$ must have memory allocation operations.
2) It or its non $f_{core}$ successors in $G$ must have string concatenate operations such as `sfprintf` or `vsfprintf`, and wildcard characters, such as a $'\%'$.
3) It or its successors in $G$ must call a $f_{output}$ to output log messages.

We implement a function named `IsCoreFunc`() to check whether a function $f$ satisfies the criterion above.

*2) Top-down slicing:* As previously discussed, a typical $f_{gen}$ passes string parameters to $f_{core}$ and is invoked at various call points. Therefore, we implement a function named `FindCallPoint`() to retrieve all the call points in CFGs to the functions with string parameters or strings constants appeared in their function abstracts. The return value is a set of tuples $\langle cp, f\rangle$ representing a call point $cp$ and the $f_{gen}$ candidate called at $cp$.

For each $f_{gen}$ candidate $f$, we need to check all paths starting from $f$ in $G$ and see if any of them reaches to a function previously tagged as $f_{core}$. Every time a $f_{core}$ (and thus a $f_{output}$) is confirmed, a log complete path $p_f$ can be thus constructed with $f$ as the starting node and the $f_{output}$ function as the end. We use $\mathcal{P}_f$ to denote the set of all log complete paths starting from $f$, and $\mathcal{CP}_f$ to denote all call points to $f$. Let $\mathcal{P}= \{\langle \mathcal{P}_f, \mathcal{CP}_f\rangle\}$ for all $f_{gen}$ candidate $f$.

Fig. 5 shows several examples of log complete paths, where dotted lines represent subpaths, solid lines represent function calls. Fig. 5(a) shows a complete path contains more than one $f_{core}$ functions. Fig. 5(b) shows a log complete path passing through $f_{core}$, whose two handler functions $f_{handler1}$ and $f_{handler2}$ are not included in this path, but can be retrieved from $f_{core}$ for further analysis. In Fig. 5(c) we have two log complete paths extracted, $f_{gen} \rightarrow f_{core1} \rightarrow f_{output}$ and $f_{gen} \rightarrow f_{core2} \rightarrow f_{output}$.

**Algorithm 1:** `FindPath`$(G, G')$

```
Input  : target binary code $G$ and its reversed graph $G'$.
Output: the set of log complete paths $\mathcal{P}$.
1  $S_{output} \leftarrow$ FindOutput($G$);
2  $\mathcal{P} \leftarrow \phi$;
3  for each $f$ in $S_{output}$ do
4      $\langle f', l \rangle \leftarrow$ nBFS($f, G'$);
5      while $f'$ is not Null and $l < \theta$ do
6          if IsCoreFun($f'$) then
7              $f'.tag \leftarrow f'.tag \cup f'_{core}$;
8          $\langle f', l \rangle \leftarrow$ nBFS($f, G'$);
9  $S_{cp} \leftarrow$ FindCallPoint($G$);
10 for each $\langle cp, f \rangle$ in $S_{cp}$ do
11     if $\mathcal{P}_f$ exists and is not empty then
12         add $cp$ to $\mathcal{CP}_f$;
13     else
14         $\mathcal{P}_f \leftarrow$ all paths $p_f$, s.t. $p_f$ starts from $f$, passes a
           function $f'$ with $f'.tag = f_{core}$ within $\theta$ steps from $f$,
           and ends with a function $f''$ with $f''.tag = f_{output}$;
15         if $\mathcal{P}_f$ is not empty then
16             $\mathcal{CP}_f \leftarrow \{cp\}$;
17             $f.tag \leftarrow f.tag \cup f_{gen}$;
18 for each $f$ with $f.tag = f_{gen}$ do
19     add $\langle \mathcal{P}_f, \mathcal{CP}_f \rangle$ to $\mathcal{P}$;
20 return $\mathcal{P}$;
```

```
sub_00404D18:                                              00404D9D   add    esp, 8h
                      ⋮                                                  ⋮
00404D34   push   "cannot initialize shttp context"        00404DA0   mov    eax, [ebp+arg_3]
00404D38   call   sub_00404D58                             00404DA4   push   eax
                      ⋮                                    00404DA8   call   ds:_sprintf
sub_00404D58:                                                            ⋮
00404D58   sub    esp, 400h;                               00404DB2   lea    eax, [ebp-400]
00404D5A   mov    eax, [ebp+arg_2]                         00404DB6   push   eax
00404D5C   call   ds:inet_ntoa                             00404DBA   call   sub_00404DC6
00404D76   mov    ebx, eax                                               ⋮
00404D7A   lea    eax, [ebp-400]                           sub_00404DC6:
00404D80   push   eax                                                    ⋮
00404D86   push   "[error][client %s]"                     00404DCA   mov    eax, [ebp+arg_2]
00404D87   push   ebx                                      00404DCD   push   eax
00404D98   call   ds:_ snprintf                            00404DD2   call   ds:writefile
```

Fig. 6: Assembly Code of the Source Code Shown in Fig. 2.

*3)* `FindPath`$(G, G')$*:* The pseudocode of the hybrid slicing algorithm is given in Algorithm 1, which takes an annotated call graph $G$ and its reversed graph $G'$.

Line 3 to 8 illustrate the bottom-up slicing procedure, conducting a BFS from each $f_{output}$ candidate $f$ in $G'$ to search for functions that satisfy the criterion of $f_{core}$ and tag them for later reference. We use the function `nBFS()` to obtain the current node $f'$ of the BFS and correspondent depth $l$ from $f$. Note that we set a threshold $\theta = 10$ as the limit of the depth of BFS to avoid path explosion. Line 10 to 17 illustrate the top-down slicing procedure, starting with the set of call points $S_{cp}$ given by `FindCallPoint()`. For each $f_{gen}$ candidate $f$, we find the set of all log complete paths starting from $f$, denoted as $\mathcal{P}_f$, and the set of all call points to $f$, denoted as $\mathcal{CP}_f$.

For example, Fig. 6 shows the assembly code of the source code shown in Fig. 2. Function `sub_00404DC6` is listed as a candidate of $f_{output}$, containing an output function named `WriteFile` at address `00404DD2`. Tracing along with the call chain, function `sub_00404D58` is suspected and tagged as a potential $f_{core}$ because it satisfies all the conditions:

1) it has a memory allocation operation at `00404D58`.
2) it has a string concatenate operation at `00404DA8` and a wildcard character, $'\%s'$ at `00404D86`.
3) it calls a $f_{output}$ at `00404DBA`

Then we started with function `sub_00404D18` for a call point at `00404D38`, which calls the $f_{core}$ candidate `sub_00404D58`. Hence, `sub_00404D58` is confirmed as both $f_{gen}$ and $f_{core}$, and thus, a log complete path is obtained.

## *C. Reconstruct Log Buffers*

Each log complete path $p_i$ in $\mathcal{P}$ is a skeleton of main log functions, for which GenLog identifies basic blocks and branches of all functions in $p_i$, and generates all possible execution paths, denoted as $ep$, using control flow graphs (CFGs) generated by IDA Pro. These paths represent possible executions of outputting logs, based on which we can reconstruct buffers that hold various terms of log messages and analyze the semantic meanings of these terms.

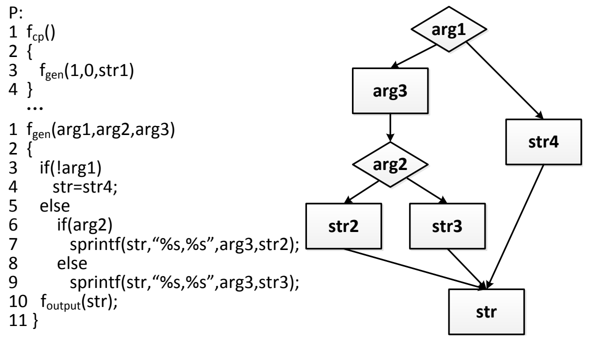
*1) Generating execution paths:* Given a log complete path $p_i$ in $\mathcal{P}$, $p_i = \langle f_1, \cdots, f_n \rangle$, GenLog finds all execution paths $ep$ from CFGs that satisfy the following two criterion: $(i)$ $ep$ must pass through $f_1, \cdots, f_n$; $(ii)$ $ep$ may include functions outside $p_i$ (for example, if an $f_{core}$ in $p_i$ calls $f_{handler}$ functions, $ep$ may pass through these $f_{handler}$ functions). Because a log complete path $p_i$ normally is a small portion of a large program, it is affordable for such analysis. Again, to avoid path explosion, we require that the calling depth of $ep$ outside of $p_i$ is no more than $\theta$.

Suppose there are $k$ execution paths found for a log complete path $p_i$, $ep_{i1}, \cdots, ep_{ik}$. For an execution path $ep_{ij}$, GenLog also records the constraints $c_{ij}$ at each control flow branch that leads to $ep_{ij}$. Later on, we will construct a log buffer $b_{ij}$ for each $ep_{ij}$, and fill in semantic tags in $b_{ij}$ using $c_{ij}$ and the parameters passed at call points to $ep_{ij}$. All $\langle c_{ij}, ep_{ij} \rangle$ tuples form a set of execution paths $\mathcal{EP}$. Note that, it is possible that GenLog cannot find any $ep$ for $p_i$, because $p_i$ is simply derived from the program's call graph without considering control flows. In this case, GenLog simply ignores $p_i$ from further analysis.

For example, Fig. 7 shows multiple execution paths for a simple log complete path $f_{gen} \rightarrow f_{output}$. If we go through different branches of the flow graph in $f_{gen}$, there are three execution paths with constraints of $[!arg1]$, $[arg1 \wedge arg2]$, and $[arg1 \wedge !arg2]$, respectively. Later on, with the parameters from the call point at line 3 of $f_{cp}$, we can finalize one log template $str1 + str3$ corresponding to this call point.

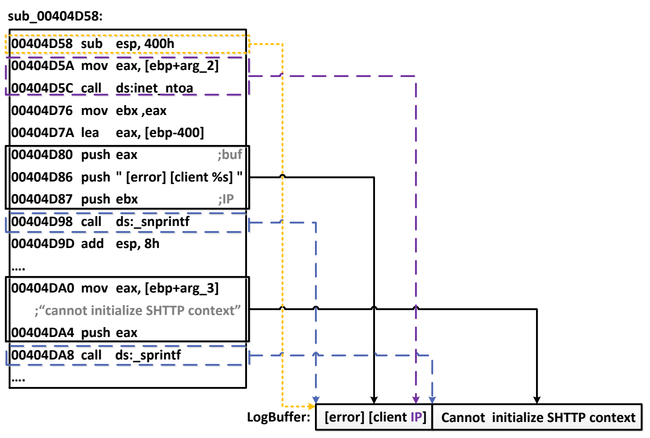
*2) Constructing log buffers:* Now we discuss how to construct log buffer $b_{ij}$ for each execution path $\langle c_{ij}, ep_{ij} \rangle$ in $\mathcal{EP}$. Let $\mathcal{B}= \{b_{ij}\}$ for all $ep_{ij} \in \mathcal{EP}$.

We identify $b_{ij}$ from the entrance of $f_{output}$ in $ep_{ij}$. Tracing back along $ep_{ij}$, we can find the address of $b_{ij}$ (for example,



```
P:
f_cp()
{
  f_gen(1,0,str1)
}
…
f_gen(arg1,arg2,arg3)
{
  if(!arg1)
    str=str4;
  else
    if(arg2)
      sprintf(str,"%s,%s",arg3,str2);
    else
      sprintf(str,"%s,%s",arg3,str3);
  f_output(str);
}
```

Fig. 7: Example of Solving Constraints.



```
sub_00404D58:
00404D58  sub   esp, 400h
00404D5A  mov   eax, [ebp+arg_2]
00404D5C  call  ds:inet_ntoa
00404D76  mov   ebx ,eax
00404D7A  lea   eax, [ebp-400]
00404D80  push  eax                        ;buf
00404D86  push  " [error] [client %s] "
00404D87  push  ebx                        ;IP
00404D98  call  ds:_snprintf
00404D9D  add   esp, 8h
....
00404DA0  mov   eax, [ebp+arg_3]
     ;"cannot initialize SHTTP context"
00404DA4  push  eax
00404DA8  call  ds:_sprintf
....
```

Fig. 8: Log Buffer Reconstruction.

the return value of `call _malloc`) and the string operations that are responsible for the content of $b_{ij}$.

A string operation $StrOp$, such as `call _sprintf`, is usually a lib function, whose interface is available for interpreting its parameters. At the call site of each string operation $StrOp_l$ in $ep_{ij}$, we can find the address of its target string $loc_l$, and its parameters $\langle arg_l 1, \cdots, arg_l n\rangle$. Tracing through $ep_{ij}$, we can reconstruct $b_{ij}$ with these string operations and their parameters in the correct order.

For example, Fig. 8 shows how the log buffer is reconstructed for the example in Fig. 2. Line "`00404D58 sub esp, 400h`" indicates the location of the log buffer. Two string operations at line `00404D98` and `00404DA8` write to this buffer with contents of "`[error][client ebx]`" and "`eax`", respectively. As a result, the obtained log buffer is "`[error][client ebx] eax`", where parameters `ebx` and `eax` will be solved later.

*3) Solving parameters:* In order to complete log templates, we need to solve parameters for each string operation $StrOp_l$ in $ep_{ij}$. We divide these parameters into two categories: *cp*-relevant parameters and *cp*-irrelevant parameters.

*cp*-relevant parameters are assigned by formal arguments of $f_{gen}$, which are passed with actual values at *cp*. We refer to the arguments of $f_{gen}$ as $arg_{gen}$, and replace these *cp*-relevant parameters with their correspondent $arg_{gen}$ in $b_{ij}$. For example, as shown in Fig. 8, by line `00404DA0` and `00404DA4`, we know `eax` in the obtained log buffer "`[error][client ebx] eax`" is $arg_{gen}3$, so we replace `eax` with $arg_{gen}3$, and the log buffer becomes "`[error][client ebx]` $arg_{gen}3$". The actual values of *cp*-relevant parameters are solved at each *cp*, which we will discuss in section III-D.

*cp*-irrelevant parameters are those independent of $f_{gen}$'s parameters, which can be a string constant, an integer constant, or a return value of a lib function with semantic information. If we can find fixed values for *cp*-irrelevant parameters by data flow analysis [23], we tag the found string constants as event terms or tokens depending on their length and put these string constants at the corresponding locations in $b_{ij}$.

The lib functions that return semantic contents usually include timestamps, filenames, hostnames, ports, IP addresses, and so on. In our approach, a semantic function table was established in advance (Table I). We adopt taint propagation analysis [16] to find whether an unknown parameter is assigned the return value of a semantic function in Table I. If so, it will be labeled with a semantic tag in $b_{ij}$ accordingly.

For example, as shown in Fig. 8, line `00404D5C` indicates an IP string will be generated by lib function `inet_ntoa`, we can replace `ebx` in the log buffer with a tag `IP`, and the log buffer becomes "`[error][client IP]` $arg_{gen}3$".

## D. Solve cp-relevant Parameters

For each call point *cp* in $\mathcal{CP}_i$ that calls an $f_{gen}$ function $f_i$, we first resolve all parameters passed at *cp* using similar techniques. Because the number of parameters passed at *cp* normally is small and sometimes they are directly passed as string contants or integer contants, this step is not very expensive.

The actual values of the parameters passed at *cp* become the values of the arguments of $f_i$, i.e., $arg_i$. We can find out which execution paths the function call of $f_i$ will follow with the actual values of $arg_i$. If $arg_i$ can uniquely determine a single execution path $ep_{ij}$ by satisfying its constaints $c_{ij}$, we can fill in the corresponding log buffer $b_{ij}$ with the actual values of $arg_i$ to obtain the final log templete. For example, in Fig. 6, we solve $arg_{gen}3$ at line `00404D34` as `"Cannot initialize SHTTP context"`. Now we obtain the final log template: `"[error][client IP] Cannot initialize SHTTP context"`.

If multiple execution paths are found due to unsolved parameters at *cp*, we obtain final log templates for all these execution paths and associate them with *cp*.

There may still exist parameters in log buffers that cannot be recognized by IDA Pro or data flow analysis [23] at this point, which usually are either passed by registers, or by indirect addresses to complex data structures. For example, a sample source code and its assembly code obtained from IDA Pro are shown in Fig. 9, where in line 96 `ebx` is a register pointing to a complex data structure.

Algorithms that analyze elements of a structure for binary code are available [5], [30]. However, the challenges are

```
12  mylog->token = "token1";            12  mov   [ebx+28h], "token1"
          ⋮                                          ⋮
28  http->log = mylog;                  56  mov   [ecx+12h], ebx
          ⋮                                          ⋮
35  http->log->token = "token2";        63  mov   [ecx+40h], "token2"
          ⋮                                          ⋮
50  log_error( mylog, "message");       95  push   "message"
                                        96  push   ebx
                                        97  call    sub_log_erro
          ⋮                                          ⋮
```

Fig. 9: Example of Complex Data Structure.

multiple aliases and assignments far away from log functions. For example, in order to trace the element `token` in the structure `mylog` in Fig. 9, these algorithms would trace all aliases of `ebx`, `ebx` $+$ `28h`, `ecx`, and `ecx` $+$ `12h`. For large binary code, such precise analysis is too expensive for our purpose.

We propose a tag-aimed trimming method to simplify alias analysis. We consider the following two conditions for early termination of alias analysis. $(i)$ An alias is found to be assigned with a string constant or integer constant, because it may be a token, a status term, or an event term, which can be used to fill in more details in log buffers. $(ii)$ An alias is labeled with a tag in Table I, which can be filled in log buffers too. For example, to sovle the parameter at line 96 in Fig. 9, we just focus on all elements of `ebx` that are assigned with constants or can be labeled with tags in Table I. Once the alias analysis reaches line 12 and finds such assignment, we record the element of `ebx` $+$ `28h` as "`token1`" and stop further analysis of its aliases, although its value might be changed by its aliases.

## IV. Experiments

We evaluated GenLog on four applications (including one web 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 | Dscription | Size |
|---|---|---|---|---|
| mkdir | 8.13 | ELF | GNU coreutils | 42KB |
| tar | 1.28 | ELF | GNU coreutils | 299KB |
| plink | 0.67 | PE | network tool | 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. We used $\theta = 10$ for all of our experiments.

### A. Log Template Evaluation

For mkdir, tar, and plink, we obtained the testing log file by executing the applications. For Nginx, we obtained the testing 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 testing log file, $L_m$ represents the number of logs whose manually extracted log templates are matched with the ones found by GenLog. We say two log templates match when they have the same event, status, variable terms, and tokens, and in the same order. $L_f$ represents the number of logs for which GenLog discovered the event term in their log templates but not with a complete match. Furthermore, we use $L_{nf}$ to represent the number of logs for which 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 testing log file, and $T_m$ represents the number of manually extracted log templates that are matched with the ones found by GenLog. $T_f$ represents the number of log templates for which GenLog discovered the event term but without 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 testing log file of mkdir, tar, and plink. For Nginx, GenLog found 46 out of 53 log templates with a perfect match, found the event term 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 | 8 | 0 | 0 | 100% | 100% |
| tar | 15 | 0 | 0 | 100% | 100% |
| plink | 20 | 0 | 0 | 100% | 100% |
| Nginx | 46 | 6 | 1 | 88.4% | 86.7% |

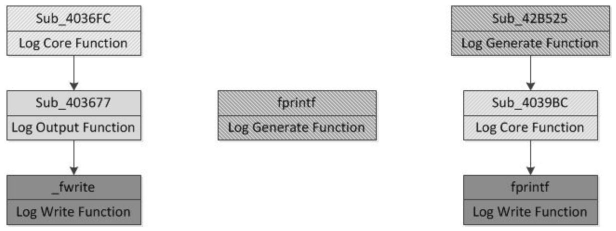


Fig. 10: Log Complete Paths of plink

## C. Case Studies

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

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 10 shows the three log complete paths found by GenLog.

TABLE VI: Prototypes of Log Generation Functions of plink

| Function | Prototype | # of Calls |
|---|---|---|
| `sub_4036FC` | $(\mathtt{void} * \mathtt{Var}, \mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 11 |
| `sub_42B525` | $(\mathtt{void} * \mathtt{Var}, \mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 152 |
| `fprintf` | $(\mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 23 |

GenLog found three log generation trees in plink as shown in Fig. 10. Table VI lists three log generation functions, their prototypes, and the number of call points, among which `fprintf` is also an $f_{core}$ function and an $f_{output}$ function.

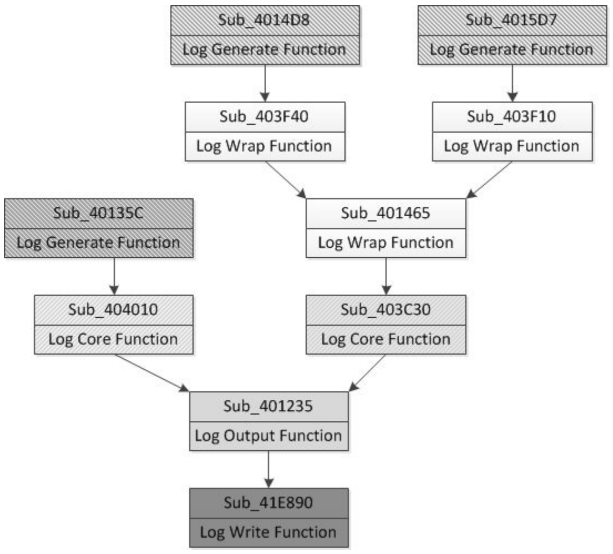


Fig. 11: Log Complete Paths of Nginx

The log templates present in the testing 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, which has a log module and complicated data structures to handle logs. Fig. 11 shows the log complete paths in the format of a tree found by GenLog. In Fig. 11, we can see various $f_{output}$ functions, $f_{core}$ functions, and $f_{gen}$ functions. We also show the three log generation functions, their prototypes, and the number of call points in Table IX.

TABLE IX: Prototypes of Log Generation Functions of Nginx

| Function | Prototype | # of Calls |
|---|---|---|
| `Sub_4014D8` | $(\mathtt{intState}, \mathtt{void} * \mathtt{Var}, \mathtt{intState}, \mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 833 |
| `Sub_4015D7` | $(\mathtt{intState}, \mathtt{void} * \mathtt{Var}, \mathtt{intState}, \mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 658 |
| `Sub_40135C` | $(\mathtt{intState}, \mathtt{constchar} * \mathtt{Format}, \mathtt{args})$ | 12 |

For Nginx, GenLog found 46 out of 53 log templates with a perfect match, found the event term 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

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 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 |

tokens, 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 IDA Pro. We can see that for small binaries GenLog can finish under 2 minutes, and almost scales linearly to larger size of binaries.

## V. Related Works

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

Yuan et al. [38] and Nguyen Hiep et al. [27] matched the log templates extracted from source code with log files to narrow down the set of the execution paths. These methods relied on source code, which may not be available in many casses. Our approach is based on stripped binaries without using log files nor source code. Executables are always at hand and easier to obtain.

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] used interprocedural 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 does not require any knowledge about the program before analysis.

Christodorescu and Kidd [12] proposed a string-analysis technique 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. They also built their analysis on an existing work called Java String Analyzer (JSA). Our approach does not need a given starting point.

Caballero et al. [33] proposed an automatic technique for extracting encryption routines from binary code. They used a dynamic approach to identify the component of web protocols. Their method adopted dynamic analysis, which can only obtain information from the instruments in execution paths of given inputs. Our approach uses static analysis that can go through all parts of binaries without designing an input.

Slicing is a commonly used code analysis technique [22], [32], [35], [36]. In our approach, we use a top-down and bottom-up slicing method to get all log-related functions in binary code, which is similar with the work of Zhang et al. [40], but with a different purpose.

## VI. Conclusion

We proposed a new approach to extract log templates from stripped C-style binaries. Our approach GenLog does not need source code nor any debugger information in the binaries. GenLog adopted a hybrid slicing method to find log related functions to reduce the scope of code analysis, and is able to discover precise log templates and their call points in the code.

## 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.