Digital Forensic Reconstruction of A Program Actions

Ahmed F. Shosha, Lee Tobin and Pavel Gladyshev
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland.
{ahmed.shosha, lee.tobin}@ucdconnect.ie, pavel.gladyshev@ucd.ie

Abstract—Forensic analysis of a suspect program is a daily challenge encounters forensic analysts and law-enforcement. It requires determining the behavior of a suspect program found in a computer system subject to investigation and attempting to reconstruct actions that have been invoked in the system. In this research paper, a forensic analysis approach for suspect programs in an executable binary form is introduced. The proposed approach aims to reconstruct high level forensic actions and approximate action arguments from low level machine instructions; That is, reconstructed actions will assist in forensic inferences of evidence and traces caused by an action invocation in a system subject to forensics investigation.

Keywords: Program Analysis, Data Flow Analysis, Digital Forensic Investigation, Action Reconstruction, Static Code Analysis.

I. INTRODUCTION

In digital investigation, investigators are required to analyze suspect executable binaries of programs found in a system subject to investigation. Program analysis, generally, can be accomplished in two-folds: (a) dynamic program analysis [1-2] (b) static program analysis [3-4]. In dynamic program analysis, a suspect binary is executed in a virtual or emulated system and actions invoked in the concrete execution (e.g. file created, registry modified, process accessed) are monitored to determine the program behavior. Concrete program execution of a program, however, has a set of limitations [5-6]. Programs are comprised of several execution paths and sets of configurations, and each path can invoke several subsequent actions. An analysis system, which a program is executed, is typically a standard and pre-configured environment that barely similar to investigated system. A program actions invoked in a concrete execution on analysis system, as a result, may be different than actions executed in a system subject to investigation. That is, forensic analysis based on concrete execution may conclude to invalid results. To supplement forensic investigation based dynamic analysis approaches, static program analysis is, then, introduced to the forensics investigation process.

Fundamentally, static program analysis approaches aim to approximate a behavior of a program if executed on a computer system. Prevalent binary analysis frameworks, (e.g. BAP [2], BitBlaze [7], or Jackstab [8]) proposed different approaches that allow automating static analysis of a vector of dependent machine instructions decoded from an executable binary. Although, these approaches allow analysis of tremendous security problems, they are however, limited if used in forensic investigation of a program’s binary. These approaches address the problems related to semantic analysis of low-level machine instructions and its side effects [9] and do not approach the analysis of a program actions that may change the final state of a system.

Forensic analysis of suspect programs, naturally, concerned with reconstruction of high level program actions (e.g. file modifications or registry manipulation) that change the final state of a system and cause traces that assists the process of evidence inferences. In previously mentioned static analysis frameworks, instructions that handle action invocation and termination, such as, call and ret are treated as basic assignment and jump operations, and arguments of an action in the procedure stack are not forensically considered in the analysis. Thus, forensic analysis based on these approaches may conclude the possibility of certain action invocation, however, the detailed specifications of the action arguments remain unspecified; i.e. a human investigator may infer, in static code analysis, the possibility of file creation based on the existence of a file create action call instruction, however, file specification cannot be determined due to the lack of action arguments analysis in the procedure block stack.

In this research, an action reconstruction approach is proposed to determine invoked actions and compute an approximation of action arguments in a procedure block. In the proposed approach, an enhancement to interprocedural analysis of a procedure block is proposed through modeling the local stack frame. Modeled stack frame is, then, augmented with a data flow analysis of action arguments to allow approximation of argument passed to an action. Determined actions and approximated action arguments values will, subsequently, allow inferences of program traces and evidence and will assist in forensic reconstruction of a program behavior in the system subject to investigation.

The remainder of the paper is organized as follows: In section Two, an intermediate language describing the semantic of a program machine code is proposed; then, an interprocedural analysis of program blocks and data flow analysis of action and arguments are presented. In section Three, the system implementation and preliminary experimental results are described. Finally, section Four, concludes the presented approach and proposes the future research work.
II. FORENSIC ANALYSIS OF A PROGRAM

A. A Program Formalization

To allow analysis of a program’s binary executable $P$, a simplified Intermediate Language (IL) [10-11] is proposed to express the concrete semantic of low level instructions belonging to a program subject to investigation. In proposed IL, a program is a set of statements $S^*$ that represent different operations over expression $e$ or a variable $v$, i.e. variable or memory assignment, conditional assessment of an expression or jump to a specific program point $\ell$. The syntax of proposed IL is presented in Figure 1. A semantic of statement $s_k$ in a program is modeled as a program state at program point $\ell$. A transition function $\delta$ model the changes in a program state $\delta(s_n)^{\ell'} \mapsto (s_n)^{\ell''}$, and updates the program counter $pc^{\ell'} := pc^{\ell''}$. The operational semantic of presented IL is shown in Figure 2. It defines unambiguously the concrete execution of an investigated program abstracted in IL. In IL operational semantic, each statement is substituted with one or more production rules that are depicted in Figure 2. All production rules are in the following form:

$$\frac{\text{State Entry Analysis}}{\text{State Post Computation}}$$

Each production rule performs analysis to a statement state before and after the concrete execution of the statement semantics. The “before” analysis is denoted as “State Entry Analysis”, where an evaluation of variable’s or expression’s state that may be effected by a statement computation in the context $(q)$ is performed. While “After” analysis, denoted as “State Post Computation”, which valuate a variable or an expression based on a statement semantic in the context of statement state $(q)$. Evaluation and valuation of a variable or expression are accomplished through proposed $\text{Test}$ and $\text{Val}$ operators, respectively.

A Program, as well, is comprised of a set of procedure blocks $P := \{p_0, p_1, \ldots, p_N\}$, such that, a procedure $p_k$ is comprised of a set of statements $s_k \in S^*$. A control flow graph (CFG) of a given $p_x$ can be constructed using a standard CFG construction technique such as presented in [3]. Note that, a program based on previously illustrated notation can be viewed as CFG of procedure blocks; where each block $p_x$ is a node in the graph, and have entry point $s_{\text{entry}}$ and exit point $s_{\text{exit}} \in S^*$ denoting the vertices. The semantic of a state transition to a procedure $p_x$, additionally to updating the program counter, allocates a memory region to the local stack frame for local variables in $p_x$ and other conventional operations such as, caller and callee saving registers [12]. Thus, in order to reason a behavior of actions invoked in $p_x$, a simplified modeling of a $p_x$ local stack frame is proposed.

A local stack frame $s$ of a procedure at determined program point is formalized as a flat lattice $s[p_x]^\ell$ that basically, computed based on statements $s_x \in [p_x]^\ell$ that semantically affect $s$. Consequently, a function $f$ is said to increment or decrement $s[p_x]^\ell$ if at any program point $\ell'$ in $p_x$, there is a statement $s_x$ that semantically affects $s[p_x]^\ell$ and $f$ is defined as, $f: \gamma[s_x], \ell' \mapsto s[p_x]^\ell$ where, $\gamma[s_x]$ is a set of data flow analysis computations over $s_x$, and it will be explained later.

B. Procedure Block Analysis

As explained, a procedure block $p_x$ is comprised of a set of statements $s \subseteq S^*$ in which, $s_{\text{entry}}$ allows in deducing $\bot$ and $s_{\text{exit}}$ deducing $\top$ in $s[p_x]^\ell$. Since a single $p_x$ may have several execution paths based on its CFG, analysis of actions invoked in $p_x$ required, primarily, identifying

![Image](https://via.placeholder.com/150)

Figure 1: A syntax of IL to Abstract a Program Semantic

![Image](https://via.placeholder.com/150)

Figure 2: Concrete Operational Semantic of IL
possible paths that hold an action. As a preliminary analysis, we formulate an execution path in $CFG[p_x]$ as a trace $t$ that is a set of transitive computation $s[x_{entry}]$ and ending at $s[x_{exit}]$. Every trace $t \in trace'$ is corresponding to a concrete execution of $p_x$, subject of analysis. Since several traces $t[p_x]$ can be computed, $s[p_x]$ may have a different layout, each corresponds to a particular executed trace in $p_x$. Thus, a stack frame of a given trace can be given as $t[s_x] := t[p_x] ightarrow s[p_x]$.

To decrease the complexity of action reconstruction from several traces found in several procedures, we restrict traces subject to analysis to those, only, hold actions (e.g. invokes OS system calls or services) and may change a system final state, if executed.

As a result, a forensic analysis of a procedure $p_x$ is only accomplished to $\cup_{stack} t_{action} \subseteq trace'$ and its $t_{action}[s_x]$.

C. Forensic Reconstruction of Actions

In digital forensics, an action is an external event to a system and action invocation may cause a creation or modification of the system objects [13-15]. Inferences and/or deduction of an action effect on a system objects in forensic investigation, required reconstruction of the action and associated specifications. To reconstruct an action from low-level instructions, a set of statements $s_x[p_x]$ in $p_x$ that invoke an action have to be determined and action arguments have to be computed or approximated.

To determine a trace hold a certain action, we defined a set of possible actions that may be invoked by a program binary and its default arguments as stated in the operating system specification [16]. The possible actions set $A$ is defined as 2-tuple of action $a_x$ and arguments $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, where, $A := \{(a_1, \alpha_1, \alpha_2, \ldots, \alpha_n), (a_2, \alpha_1, \alpha_2, \ldots, \alpha_n), \ldots\}$. For every trace $t \subseteq trace'$, if $s_x \in s_x[p_x][t]$ invokes an action $a_x \in A$; $t$ is, then, labeled as $t_{action}$ and $t$ at $s_x$ is labeled for further analysis.

As shown in figure 3, a code portion of a trace from a procedure block of a malicious program invoke actions at program points 54 and 7C. Action at 54 accesses a file in the system, while action in 7C creates a persistent service in the system which leaves a trace that may assist a forensic investigation. Determining an action may invoke at a certain procedure block in a concrete execution, however, may not completely assist an inference of forensic evidence unless the arguments to the action are specified, as well. For example, action at program point 7C can assist in inference of system service creation, however, service specification (e.g. name, desired access, path to a service binary image) still unspecified. The service name of action at 7C is added to $s[p_x]$ at $t_7C$, however, the value of a register variable $eax$ (an action argument register) has been defined through several computations prior to stack decrementing at $t_71$. To compute or approximate action arguments, a data flow analysis of arguments in $s[p_x][t]$ is proposed.

**Figure 3: A Portion of a Trace Code from a Malicious Program**

A standard data flow analysis technique denoted as variable assignment definition [3] is employed to determine a set of statements $s_x[p_x][t]$ that previously, assigned a value to an argument register used in a subsequent action invocation. Let $R := \{r_1, \ldots, r_{m}\}$ denote a set of register variables which $s_x[p_x][t]$ operates on. Let $\gamma := \{(r_1, \ell_1), \ldots, (r_n, \ell_n)\}$ a Poset of 2-tuple, $\ell$ of $s_x$ in $p_x$ where $s_x[p_x][t]$ has found to valuate $r_{\ell}$. An assignment analysis of variables used in subsequent invocation of an action can be defined as a backward trace function $f^{-1}$ over $s_x[p_x][t]$ to trace values assigned to a variable of interest from $s_{action}[t]$ to $s_{entry}[p_x][t]$, such that:

$$\gamma := f^{-1}: s_{action}[p_x][t] \mapsto s_{entry}[p_x][t]$$

The set $\gamma$ resulted is, then, mapped to an argument of action at $t[s_x]$, as follow:

$$f: (\gamma[s_x], \ell) \mapsto t[s_x]$$

Finally, for every $s_x \in \gamma[s_x]$, a concrete evaluation using $\text{Val}$ and $\text{Test}$ is accomplished to compute a concrete value for action arguments subsequently computed though several statements determined in $\gamma[s_x]$, as follow:

$$\forall s_x \in \gamma[s_x]$$

$$f: (\text{Test } s_x \land \text{Val } s_x) \mapsto v^a$$

As shown in Figure 3, an argument of action specified at 7C is added $s[p_x][t]$ at $t_71$. A backward trace function over action argument at $s_{action}[p_x][t_71]$ back to $s_{entry}[p_x][t]$ is recursively invoked to determine $s_x \in t[s_x]$ that assigned a value to variables used as arguments to action $s_{action}$, i.e. $\gamma[s_{\ell^1}] := \{(eax, 60), (eax, 5E), (eax, 5A)\}$. 

121
Test and Val operators are, then, recursively operate over $\gamma$ to compute and valuate an argument variable, i.e., $\epsilon \alpha x$ defined in $\gamma$ set.

### III. IMPLEMENTATION AND PRELIMINARY RESULTS

A prototype program code disassembler and analyzer is implemented to presented program forensics approach. Developed prototype automates x86 executable binary forensic investigation through decoding a suspect binary machine-code and lift it to our proposed IL. Lifted instructions are, then, automatically examined with the proposed action reconstruction and arguments computing algorithms, as described.

To evaluate the proposed approach, forensics analysis is performed for different samples of malicious programs used to commit cybercrimes, such as variants of Zeus (a malware family for banking cybercrimes) and other ransom malware program [17]. Every sample subject to investigation has been forensically examined in our developed prototype and concretely executed in managed system to evaluate the preciseness of reconstructed actions relative to concrete execution, and to determine whether all action arguments have been successfully computed. A sample preliminary result of presented approach is depicted in Table 1. As shown in Table 1, a several forensic actions and associated arguments have been, successfully, reconstructed from machine code, and action traces have been located in the system subject to investigation. The accuracy percentage describes the percentage of successfully reconstructed actions arguments values in compare to concrete execution. In illustrated experimental results, the proposed approach has successfully reconstructs a considerable set of actions that have invoked in a concrete execution and computed substantial percentage of action arguments values; however, a set of action arguments have not been computed, since the behavior of samples subject to investigation are developed to execute based on runtime dynamic computation and configuration parameters in the compromised systems. In other words, several reconstructed actions in the samples are operated on arguments that dynamically populated in runtime from compromised systems, and hence, to approximate such values, a detailed modeling of compromised system is required to be included in the forensic analysis process.

<table>
<thead>
<tr>
<th>Sample Name</th>
<th>#IL</th>
<th>#Actions</th>
<th>Accuracy %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan.Zbot-1225</td>
<td>1031</td>
<td>34</td>
<td>89</td>
</tr>
<tr>
<td>Trojan.Zbot-385</td>
<td>1319</td>
<td>53</td>
<td>87</td>
</tr>
<tr>
<td>Trojan.Zbot-1023</td>
<td>1220</td>
<td>39</td>
<td>82</td>
</tr>
<tr>
<td>Trojan.Zbot-1652</td>
<td>2240</td>
<td>103</td>
<td>71</td>
</tr>
<tr>
<td>Trojan.Ransomwre-1</td>
<td>1632</td>
<td>89</td>
<td>79</td>
</tr>
</tbody>
</table>

Table 1: A Sample Forensic Analysis of Malicious Programs

### IV. CONCLUSION

In this research work, an automated approach to extract forensic actions from low-level machine code and approximating action arguments values based on backward data flow analysis algorithm is proposed. The proposed approach allows in inference/deduction of evidence and extraction of traces related to a suspect program in a system subject to forensics investigation.

A prototype forensics framework is, then, developed and evaluated using different malicious programs that regularly used to commit cybercrime activities.

### REFERENCES