2013 IEEE Security and Privacy Workshops

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 by 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 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 in 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.

DOI 10.1109/SPW.2013.17

```
<Statement 𝕊 >       ::= < expression >
                     |<variable> ::= < expression >
                     |if < expression >: <statement>
                     else <statement>
                     |<jump>: < statement >

<Expression 𝕖 >      ::= <variable>
                     |<assignment >
                     |<test> <expression>
                     |<valuation> <expression>
                     |<unary-expression>
                     |<binary-expression>
                     | <value>
```

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

II. Forensic Analysis of A program

A. A Program Formalization

To allow analysis of a program's binary executable $\mathbb{P}$, 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 $\mathbb{S}^*$ that represent different operations over expression $\mathbb{e}$ or a variable v, i.e. variable or memory assignment, conditional assessment of an expression or jump to a specific program point ℓ. The syntax of proposed IL is presented in Figure 1. A semantic of statement $\mathbb{s}_k$ in a program is modeled as a program state at program point ℓ. A transition function δ model the changes in a program state $\delta\langle \mathbb{s}_m \rangle^{\ell} \longmapsto \langle \mathbb{s}_n \rangle^{\ell'}$, 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:

$$\textit{Operational semantic rule} \ := \ \frac{\textit{State Entry Analysis}}{\textit{State Post Computation}}$$

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 $(\mathscr{q})$ 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 $(\mathscr{q})$. Evaluation and valuation of a variable or expression are accomplished through proposed $\widehat{\text{Test}}$ and $\widehat{\text{Val}}$ operators, respectively.

A Program, as well, is comprised of a set of procedure blocks $\mathbb{PB} := \{\mathscr{p}_0, \mathscr{p}_1, .., \mathscr{p}_x\}$, such that, a procedure $\mathscr{p}_k$ is comprised of a set of statements $\mathbb{s}_l \in \mathbb{S}^*$. A control flow graph (CFG) of a given $\mathscr{p}_k$ 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 $\mathscr{p}_x$ is a node in the graph, and have entry point $\mathbb{s}_{entry}$ and exit point $\mathbb{s}_{exit} \in \mathbb{S}^*$ denoting the graph vertices. The semantic of a state transition to a procedure $\mathscr{p}_x$, additionally to updating the program counter, allocates a memory region to the local stack frame for local variables in $\mathscr{p}_x$ and other conventional operations such as, caller and callee saving registers [12]. Thus, in order to reason a behavior of actions invoked in $\mathscr{p}_x$, a simplified modeling of a $\mathscr{p}_x$ local stack frame is proposed.

<Assignment $\mathcal{V}$ >:

$$:= \frac{\widehat{\text{Test}}: \mathcal{V} \ \longmapsto (\mathscr{q})}{\widehat{\text{Val}}: \mathcal{V}\ (\mathscr{q}) \longmapsto [\![\widehat{\text{Test}}: e \longmapsto (\mathscr{q})]\!]\ ;\ \delta\ (\mathscr{q}\ , \ell) \longmapsto (\mathscr{q}', \ell')}$$

<Conditional Expression>:

$$:= \frac{\widehat{\text{Test}}: e \ \longmapsto (\mathscr{q})}{\widehat{\text{Val}}: \mathscr{q} \longmapsto \ [\![\widehat{\text{Test}}: e \longmapsto (\mathscr{q})]\!] \text{ OR } \delta\ (\mathscr{q}_{\epsilon)}\ ;\ \delta\ (\mathscr{q}\ , \ell) \longmapsto (\mathscr{q}^?,\ \ell^?)}$$

<Jump>:

$$:= \frac{\widehat{\text{Test}}: e \ \longmapsto (\mathscr{q})}{\delta\ (\mathscr{q}\ , \ell) \longmapsto (\mathscr{q}',\ \ell')}$$

$\widehat{\text{Test}}$ Operator:

$$:= \frac{\text{Temp}: \tau}{\tau \ \longmapsto [\![\text{Compute}: e \ \longmapsto (\mathscr{q})]\!]}$$

$\widehat{\text{Val}}$ Operator:

$$:= \frac{\text{Temp}: \tau \ \longmapsto \ [\![\widehat{\text{Test}}: e \longmapsto (\mathscr{q})]\!]}{\mathcal{V} \ := \ \tau}$$

Figure 2: Concrete Operational Semantic of IL

A local stack frame $\mathscr{s}$ of a procedure at determined program point is formalized as a flat lattice $\langle \mathscr{s}[\![\mathscr{p}_y]\!]^{\ell}, \sqsubseteq \rangle$. $\bot \in \mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ denotes an empty stack frame and $\top \in \mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ denotes the top of a stack. The size of a $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ is, basically, computed based on statements $\mathbb{s}_z \in [\![\mathscr{p}_y]\!]^{\ell}$ that semantically affect $\mathscr{s}$. Consequently, a function f is said to increment or decrement $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ if at any program point ℓ' in $\mathscr{p}_x$, there is a statement $\mathbb{s}_z$ that semantically affects $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ and f is define as, $f\colon \langle \gamma[\![\mathbb{s}_z]\!], \ell' \rangle \longmapsto \mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ where, $\gamma[\![\mathbb{s}_z]\!]$ is a set of data flow analysis computations over $\mathbb{s}_z$, and it will be explained later.

B. Procedure Block Analysis

As explained, a procedure block $\mathscr{p}_x$ is comprised of a set of statements $\mathbb{s} \sqsubseteq \mathbb{S}^*$ in which, $\mathbb{s}_{entry}$ allows in deducing $\bot$ and $\mathbb{s}_{exit}$ deducing $\top$ in $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$. Since a single $\mathscr{p}_x$ may have several execution paths based on its CFG, analysis of actions invoked in $\mathscr{p}_x$ required, primarily, identifying possible paths that hold an action. As a preliminary analysis, we formulate an execution path in CFG$[\![\mathscr{p}_x]\!]^{\ell}$ as a *trace* $\mathscr{t}$ that is a set of transitive statements starting at $\mathbb{s}_{entry}$ and ending at $\mathbb{s}_{exit}$. Every trace $\mathscr{t} \in trace^{*}$ is corresponding to a concrete execution of $\mathscr{p}_x$ subject of analysis. Since several traces $\mathscr{t}[\![\mathscr{p}_x]\!]$ can be computed, $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ may have a different layouts, each corresponds to a particular executed trace in $\mathscr{p}_x$. Thus, a stack frame of a given trace can be given as $\mathscr{t}[\![\mathscr{s}_x]\!] := \mathscr{t}[\![\mathscr{p}_x]\!] \longmapsto \mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$.

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 $\mathscr{p}_x$ is only accomplished to $\mathscr{t}_{action} \sqsubseteq trace^{*}$ and its $\mathscr{t}_{action}[\![\mathscr{s}]\!]$.

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 $\mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ in $\mathscr{p}_x$ 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 $\mathbb{A}$ is defined as 2-tuple of action a_x and arguments $\langle \imath_1, \imath_2, .., \imath_n \rangle$, where, $\mathbb{A} := \{(a_1, \langle \imath_1, \imath_2, .., \imath_n \rangle), .., (a_k, \langle \imath_1, \imath_2, .., \imath_m \rangle)\}$. For every trace $\mathscr{t} \sqsubseteq trace^{*}$, if $\mathbb{s}_z \in \mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ invokes an action $a_1 \in \mathbb{A}$; $\mathscr{t}$ is, then, labeled as $\mathscr{t}_{action}$ and ℓ at $\mathbb{s}_z$ 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 $\mathscr{s}[\![\mathscr{p}_x]\!]^{71}$ at ℓ^{71}, however, the value of a register variable `eax` (an action argument register) has been defined through several computations prior to stack decrementing at `71`. To compute or approximate action arguments, a data flow analysis of arguments in $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$is proposed.

```
48: lea eax, [esp+1Ch]
4C: push 3E8h
51: push eax           #lpFileName
52: push 0
54: call ds:GetModuleFileNameA


  …..


62: lea ecx, [esp+414h+BinaryPathName]
66: push 0             #lpLoadOrderGroup
68: push ecx           #lpBinaryPathName
69: push 0             #dwErrorControl
6B: push 2             #dwStartType
6D: push 10h           #dwServiceType
6F: push 2             #dwDesiredAccess
71: push eax           #ServiceName
7B: push esi           #hSCManager
7C: call ds:CreateServiceA
```

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 $\mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ that previously, assigned a value to an argument register used in a subsequent action invocation.
Let $\mathbb{R} := \{\mathscr{r}_1, \dots \mathscr{r}_x\}$ denote a set of register variables which $\mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ operates on. Let $\gamma := \{\langle \mathscr{r}_1, \ell \rangle, \dots \langle \mathscr{r}_x, \ell' \rangle\}$ a Poset of 2-tuple, ℓ^{*} of $\mathbb{s}_x$ in $\mathscr{p}_x$where $\mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ has found to valuate $\mathscr{r}_x$.
An assignment analysis of variables used in subsequent invocation of an action can be defined as a backward trace function f^{-1} over $\mathbb{s}_x[\![\mathscr{p}_x]\!]^{\ell}$ to trace values assigned to a variable of interest from $\mathbb{s}_{action}{}^{\ell}$ to $\mathbb{s}_{entry}[\![\mathscr{p}_x]\!]^{\ell}$, such that:

$$\gamma := f^{-1}\colon \mathbb{s}_{action}[\![\mathscr{p}_x]\!]^{\ell'} \longmapsto \mathbb{s}_{entry}[\![\mathscr{p}_x]\!]^{\ell}$$

The set γ resulted is, then, mapped to an argument of action at $\mathscr{t}[\![\mathscr{s}_x]\!]$, as follow:

$$f\colon \langle \gamma[\![\mathbb{s}_z]\!], \ell' \rangle \longmapsto \mathscr{t}[\![\mathscr{s}_x]\!],$$

Finally, for every $\mathbb{s}_k \in \gamma[\![\mathbb{s}_z]\!]$, a concrete evaluation using $\widehat{\mathrm{Val}}$ and $\widehat{\mathrm{Test}}$ is accomplished to compute a concrete value for action arguments subsequently computed though several statements determined in $\gamma[\![\mathbb{s}_z]\!]$, as follow:

$$\forall \mathbb{s}_k \in \gamma[\![\mathbb{s}_z]\!]$$

$$f\colon \langle \widehat{\mathrm{Test}}\, \mathbb{s}_k \wedge \widehat{\mathrm{Val}}\, \mathbb{s}_k \rangle \longmapsto v^{a}$$

As shown in Figure 3, an argument of action specified at `7C` is added $\mathscr{s}[\![\mathscr{p}_x]\!]^{\ell}$ at ℓ^{71}. A backward trace function over action argument at $\mathbb{s}_{action}[\![\mathscr{p}_x]\!]^{71}$ back to $\mathbb{s}_{entry}[\![\mathscr{p}_x]\!]^{\ell}$ is recursively invoked to determine $\mathscr{s}_k \in \mathscr{t}[\![\mathscr{s}_x]\!]$ that assigned a value to variables used as arguments to action $\mathbb{s}_{action}$, (i.e. $\gamma[\![\mathbb{s}_{71}]\!] := \{\langle \mathtt{eax}, \mathtt{60} \rangle, \langle \mathtt{eax}, \mathtt{5E} \rangle .. \langle \mathtt{eax}, \mathtt{5A} \rangle\}$).

Sample Name	#IL	#Actions	Accuracy %
Trojan.Zbot-1225	1031	34	89
Trojan.Zbot-385	1319	53	87
Trojan.Zbot-1023	1220	39	82
Trojan.Zbot-1652	2240	103	71
Trojan.Ransomwre-1	1632	89	79

Table 1: A Sample Forensic Analysis of Malicious Programs

$\widehat{\text{Test}}$ and $\widehat{\text{Val}}$ operators are, then, recursively operate over γ to compute and valuate an argument variable, i.e. `eax` defined in γ 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 run-time 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.

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 forensic analysis framework is, then, developed and evaluated using different malicious programs that regularly used to commit cybercrime activities.

References

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools”, ACM Computing Surveys, vol. 44, no. 2, pp. 1–42, Feb. 2012.

[2] T. A. Edward J. Schwartz, “All you Ever wanted to know about dynamic taint analysis and forward symbolic execution”, IEEE Symposium on Security and Privacy, pp. 317–331, 2010.

[3] C. Nielson, F., Nielson, R., & Hankin, “Principles of program analysis. Springer”, p. 450, 1999.

[4] M. Christodorescu and S. Jha, “Static analysis of executables to detect malicious patterns”, 12th conf. on USENIX Security Symposium - Vol. 12, P. 12, 2003.

[5] C. Malin, E. Casey, J. Aquilina,“Malware forensics: investigating and analyzing malicious code”, Syngress, 2008.

[6] M. Brand, C. Valli, and A. Woodward, “Malware forensics: discovery of the intent of deception”, in Australian Digital Forensics Conference, 2010.

[7] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: a new approach to computer security via binary analysis”, in Intl, Conf. on Information Systems Security, vol. 5352, pp. 1–25, 2008.

[8] J. Kinder, F. Zuleger, and H. Veith, “An abstract interpretation-based framework for control flow reconstruction from binaries” in 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation, vol. 5403, pp. 214–228, 2009.

[9] A. Flexeder, M. Petter, and H. Seidl, “Side-effect analysis of assembly code”, in 18th Intl Conf. on Static Analysis, pp. 77–94, 2011.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Introduction to automata theory, languages, and computation”, Prentice Hall, p. 750, 2006.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, “Compilers: principles, techniques, and tools”, (2nd Edition). Addison Wesley, p. 1000. 2006.

[12] Intel, “Intel IA-32 Architectures Software Developer Manuals.” [Online].www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. [Accessed: 14-Feb-2013].

[13] J. James, P. Gladyshev, and Y. Zhu, “Analysis of evidence using formal event reconstruction”, Digital Forensics and Cyber Crimes, vol. 31, no. 1, pp. 85–98, 2010.

[14] P. Gladyshev and A. Patel, “Finite state machine approach to digital event reconstruction”, Digital Investigation, vol. 1, no. 2, 2004.

[15] F. A. Shosha, J. James, and P. Gladyshev, “Towards automated forensic event reconstruction of malicious Code”, 15th Intl. Symposium on Research in Attacks and Intrusion RAID, p. 388 2012.

[16] Microsoft, “MSDN: The Microsoft Developer Network.” [Online]. Available: http://msdn.microsoft.com/en-US/.

[17] “Open Malware: community malicious code research and analysis”. Available: http://www.offensivecomputing.net/.