A Generic Binary Analysis Method for Malware

Tomonori Izumida[1,2], Kokichi Futatsugi[2], and Akira Mori[1]

[1] National Institute of Advanced Industrial Science and Technology
2-3-26 Aomi, Koto-ku, Tokyo 135-0064, Japan
{tomonori.izumida,a-mori}@aist.go.jp
[2] Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{tizmd,kokichi}@jaist.ac.jp

Abstract. In this paper, we present a novel binary analysis method for malware which combines static and dynamic techniques. In the static phase, the target address of each indirect jump is resolved using backward analysis on static single assignment form of binary code. In the dynamic phase, those target addresses that are not statically resolved are recovered by way of emulation. The method is generic in the sense that it can reveal control flows of self-extracting/obfuscated code without requiring special assumptions on executables such as compliance with standard compiler models, which is requisite for the conventional methods of static binary analysis but does not hold for many malware samples. Case studies on real-world malware examples are presented to demonstrate the effectiveness of our method.

1 Introduction

Malicious software, or malware, has become a main threat to the network-centric society today. Malware scanners are used to protect computers from the threat. However, such tools rely on syntactic signature matching algorithms and are evadable by program transformation techniques. As a result, the threat is still an ongoing issue and amount of newly found malware samples are reported to anti-malware companies each day. After receiving such samples, malware analysts examine each suspicious executable to understand its behavior and effects on the system, characterize the hazard which it may cause, and extract a distinctive signature pattern from the binary. These analysis steps involve many manual operations and are time consuming.

Various methods have been proposed to automate analysis of malware binaries. Dynamic analysis, which executes a target on a debugger or an emulator within a controlled environment to observe its activity, is used in many practical studies. However, the method fails to detect fatal activities of interest if the target changes its behavior depending on trigger conditions such as existence of a specific file as only a single execution path may be examined for each attempt. The shortcomings come from the lack of global views of the program behaviors. Static analysis, on the other hand, extracts such high-level information as control

I. Echizen, N. Kunihiro, and R. Sasaki (Eds.): IWSEC 2010, LNCS 6434, pp. 199–216, 2010.

flow graphs (CFGs) from a binary and permits multi-path analysis without executing it. Enhancing the range of static analysis will greatly benefit the analysis of malware behaviors.

Static analysis may be thwarted by various code obfuscation techniques. For example, *opaque constants* [14] obtained by lengthy arithmetic or intensional errors at run-time are difficult to predict statically. A number of techniques for obstructing disassemblers using opaque constants have been reported [14]. Executable packers such as `UPX` and `ASPack` are other obstacles to static analysis, which compress executable files and prefix small decompressing routines to the compressed data for run-time extraction. Although such packers can be applied for benign software to save disk space or to reduce loading time, many malware samples also abuse executable packers to hide themselves from reverse engineering. Considering the fact that these obfuscation techniques do not pose serious problems to dynamic analysis, it is natural to combine static and dynamic methods to mutually cover their weakness. If the static analyzer is ineffective for exploring fragments of the target binary, it can rely on its dynamic counterpart.

The challenge of static binary analysis can be summarized to how we determine destination addresses of indirect jumps. For example, to follow the control flow after an instruction `CALL EAX`, we need to know what constant value will appear in the register `EAX` at the time of execution. This is a difficult task since it must be done without running the code and traces of values stored in machine registers or memory locations can be complex in malware because of its anti-analysis capabilities. It is an undecidable problem in general and restrictions and/or approximations must be introduced for a concrete solution. Current approaches for indirect jump resolution assume that that the target binary is built from high level language like C/C++ with standard compiler conventions. There are many malware samples that are written directly in assembler languages and do not conform to such conventions at all.

In this paper, we introduce a novel analysis method for malware binaries which does not rely on special assumptions on target binaries. The method combines static and dynamic analysis methods to reveal multi-path program behaviors by backward tracing of values in registers and memory locations at control transfer points. For this, a demand-driven symbolic evaluation method is developed for the static single assignment (SSA) form of machine instructions.

We present a new static analysis tool called Syman with a light-weight emulator Alligator [12,11] both implemented for binary executables for MS Windows Operation Systems on IA-32 architecture. Syman and Alligator are designed to cooperate with each other to discover entire control flows of the target. Syman works on execution paths that are not explored by Alligator while Alligator handles self-extracting code and gathers run-time information including opaque values needed by Syman.

The tools are controlled in the following manner. Syman is invoked first from the entry point of the program to build an initial CFG by resolving indeterminate destination addresses as mush as possible. Then, Alligator takes over control to start emulation from the entry point until it encounters a code block that is not

yet visited by Syman. When such disagreement occurs, Syman resumes expanding CFG from the newly found code block. In this way, Syman and Alligator work together to build an entire CFG. In a particular case where the target executable is compressed by a simple executable packer, the code block extracted by Alligator is considered unvisited and Syman proceeds to the now uncompressed part that was hidden in the original executable.

The organization of the paper is as follows. Section 2 gives an overview of the binary analysis method and explains how the method combines static and dynamic techniques. Section 3 explains the static analysis method which allows to resolve indirect jumps without making assumptions on the target executable and Section 4 touches upon issues concerning implementation. Case studies with several real-world malware samples are explained in Section 5. We finally list related works in Section 6 and concludes with future work in Section 7.

2 Method Overview

In this section, we explain ingredients of our method, the static analyser Syman and the dynamic analyzer (emulator) Alligator, and how these two interact with each other.

2.1 Dynamic Analyser

The dynamic analyser (emulator) Alligator [12,11] consists of an Intel IA32 CPU simulator, virtual memory, and a number of stub functions corresponding to Win32 API functions for emulating operating system capabilities such as process/thread management and memory management. Alligator is lightweight in the sense that it does not require an instance of Windows OS and uses dummy software objects instead of emulating actual hardware such as disk drives and network devices.

When a target executable is given, Alligator loads it into virtual memory, creates process/thread information structures in the same way as an actual Windows OS does, and loads dynamic linkable libraries (DLLs). Since Alligator emulates an OS capabilities by way of stub functions rather than executing API function code in DLLs, it does not require real DLLs distributed with Windows OS. Pseudo DLLs containing minimal information of API functions such as function names and entry addresses are used instead. Alligator notifies Syman of the lists of loaded DLLs and exported API functions when it finishes target loading. Syman uses this information to build special basic blocks corresponding to API functions (described in Section 4.1).

The OS specific control such as exception handling, thread management, and memory management is emulated accordingly in Alligator. To interpret API functions symbolically, a stub function is invoked when control is transferred to the address of the API function in virtual memory. The default task of a stub function is to remove arguments placed on top of the stack and to store a tentative return value in register EAX. To make stub calls transparent and extendable,

a machine generated pseudo library (DLL) files that imitate standard memory layout of Windows operating systems are prepared. In the current version, Alligator has approximately 1500 non-trivial stub functions over 1700 pseudo DLL files and operates without any proprietary code, which makes the system highly portable and manageable. The stub mechanism to bypasses external library code contributes greatly in making up for the processing speed. Note that Alligator uses a CPU simulator and is much slower than commercial emulators based on native execution.

Alligator has an ability to detect execution of self-modified code by using memory page management system of the IA-32 architecture. Alligator clears the dirty bit of each page table entry before execution and for each step-wise execution it checks whether the dirty bit of the current code page is set or not. If it is set, it indicates that the instruction which is going to be executed might have been modified during previous execution. Alligator flags this event to let Syman analyze the code from the address in the later stage. Self-modified code generated by advanced packers can be analyzed by Syman using this mechanism.

Alligator was originally developed for detecting unknown computer viruses/worms by identifying common malicious behaviors such as mass mailing, registry overwrites, process scanning, external execution, and dubious Internet connection by monitoring API calls during emulation. Since Alligator operates in a self-contained software process, there is no need of restarting a guest OS for a new session nor additional costs for parallel processing in multiple instances, as opposed to other methods involving full OS emulation. Alligator can be used for server-side malware detection and in fact, it had successfully detected nearly 95% of email attached malware on a working email server from 2004 through 2006 without relying on signatures of previously captured samples.

2.2 Static Analyser

From an given address in code, the static analyser Syman disassembles the code stored in the virtual memory of Alligator by recursively traversing each direct jump and translates each disassembled instruction to a sequence of statements in intermediate representations (IR) that captures operational semantics of each instruction. See Fig. 1.

Each IR statement is in the form of assignment, where a variable symbol is placed on the left hand side and an IR expression on the right hand side. A statement is said to *define* a variable v if v occurs in the left hand side, and is said to *use* v if v occurs in the right hand side. $var_{sz} \leftarrow expr_{sz}$ represents sz-bit data transfers, where var_{sz} is a variable symbol and $expr_{sz}$ is a value expression. We may omit subscripts $_{sz}$ when they are clear from the context. A *control statement* $\leftarrow$ ControlExpr defines no variable and represents control transfers.

To express the effect of a memory write operation, a *memory expression* is introduced. Assignments involving memory expressions are restricted to the form $M' \leftarrow \mathrm{St}_{sz}(M, addr_{32}, expr_{sz})$, which denotes that an sz-bit value $expr_{32}$ is stored into address $addr_{32}$ and the memory state is updated from M to M'.

$$
\begin{aligned}
\mathrm{expr}_{sz} ::= {} & \mathrm{int}_{sz} \mid \mathrm{var}_{sz} \mid \mathrm{expr}_{sz} + \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} - \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} * \mathrm{expr}_{sz} \ldots \\
& \mid \mathrm{expr}_{sz} \& \mathrm{expr}_{sz} \mid \mathrm{expr}_{sz} \| \mathrm{expr}_{sz} \mid \sim \mathrm{expr}_{sz} \ldots \\
& \mid \mathrm{Ld}_{sz}(M, expr_{32}) \\
\mathrm{ControlExpr} ::= {} & \mathrm{Jump}(\mathrm{expr}_{32}) \mid \mathrm{CALL}(\mathrm{expr}_{32}) \mid \mathrm{RET}(\mathrm{expr}_{32}) \mid \mathrm{Branch}(\mathrm{expr}_1, \mathrm{expr}_{32}, \mathrm{expr}_{32}) \ldots \\
\mathrm{MemoryExpr} ::= {} & M \mid \mathrm{St}(M, \mathrm{expr}_{32}, \mathrm{expr}_{sz})
\end{aligned}
$$

Fig. 1. Intermediate Representation

X86 Instruction	Translated Statements
`ADD EAX, 1234[EDX,ESI]`	$src \leftarrow \mathrm{Ld}_{32}(\mathrm{M}, \mathrm{EDX} + \mathrm{ESI} + 1234)$ $\mathrm{EAX} \leftarrow \mathrm{EAX} + src$
`CALL EBX`	$dest \leftarrow \mathrm{EBX}$ $\mathrm{ESP} \leftarrow \mathrm{ESP} - 4$ $\mathrm{M} \leftarrow \mathrm{St}_{32}(\mathrm{M}, \mathrm{ESP}, NextIP)$ $\leftarrow \mathrm{Call}(dest)$
`RET`	$\mathrm{ESP} \leftarrow \mathrm{ESP} + 4$ $dest \leftarrow \mathrm{Ld}_{32}(\mathrm{M}, \mathrm{ESP})$ $\leftarrow \mathrm{Ret}(dest)$

Fig. 2. Translation of Machine Instructions

Related to this is an expression $\mathrm{Ld}_{sz}(M, addr_{32})$, denoting an sz-bit value stored at address $addr_{32}$ in the memory state M. Note that a memory expression M has no bit size type.

Figure 2 shows examples of translation. Notice that indirect memory operands such as `1234[EDX,ESI]` and stack operations implied by `CALL`/`RET` are translated using memory expressions. The last statements appearing in the translation of `CALL` and `RET` are inserted for specifying the destination. The operators labeled Call/Ret are used for notational convenience.

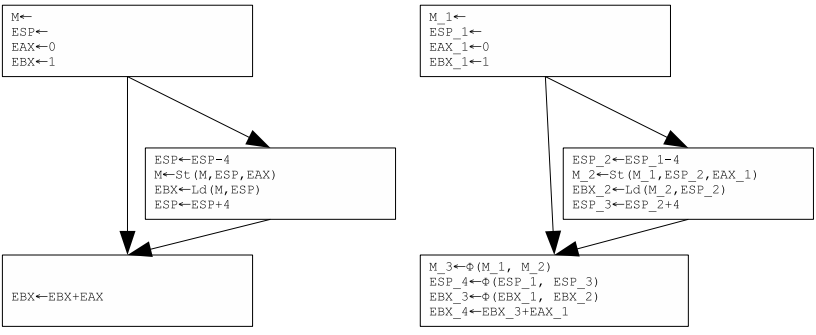
The static single assignment (SSA) form [8] is intermediate representation originally used for compiler optimization. It is obtained by converting a control flow graph and satisfies following conditions:

1. each usage of a variable is dominated by its definition, and
2. every variable is defined only once.

A control point A *dominates* another point B if A precedes B in any execution path. To meet the first condition, an assignment statement in the form $v \leftarrow \Phi(v, \ldots, v)$ is inserted at each control point where multiple definitions of a variable v merge. The function Φ on the left hand side is a pseudo function and has no algebraic meaning. Each variable symbol is renamed to satisfy the second requirement after the insertion of Φ-functions.

For example, the definition of `EAX` dominates its use in the bottom block and other variables have multiple definitions (Fig.3(a)). To convert the graph into the SSA form, Φ-functions are inserted in the bottom block for the variables except `EAX` and each variable name is renamed to make the definition unique (Fig.3(b)).

Syman continues instruction translation illustrated above until it encounters a control transfer instruction. If its destination address is immediately determined,



(a) Original CFG (b) Converted SSA Form

Fig. 3. Example of SSA Form

a new basic block is created from the address. Otherwise, the node is marked as indeterminate. After all immediate addresses have been processed, the control flow graph is converted into the SSA form, in which indeterminate destinations of marked nodes are examined if they evaluate to an immediate value. This evaluation step takes an *demand-driven* approach [18,20], in which variables in the evaluated expression is replaced with their definitions by tracing their def-use chains. When evaluating an expression $\mathrm{Ld}_{sz}(M, addr)$, *memory state tracing* is performed by searching the most recent expression of the form $M'' \leftarrow \mathrm{St}_{sz}(M', addr', val)$ where $addr$ and $addr'$ coincide. The expression will be replaced with value val.

The process can be complex when it involves a code segment that is reached from/returns to multiple points since the calling context must be identified to trace values in def-use chains of variables. This is what is called "inter-procedural analysis" in a traditional setting.

For normal executables generated by standard compilers, it can be done by identifying subroutines/procedures and replicating them for each call site. This is not difficult because a subroutine is entered at the same address and returns to the next instruction of the calling `CALL` instruction by the `RET` instruction at the bottom. However, in malware code generated by an assembler, code sharing can be hidden in most obscure ways. For example, inserting an irregular `RET` instruction after pushing an arbitrary value on the stack can cause static analysis a serious trouble.

Static analysis of malware binary code thus requires a new method for generalized inter-procedural analysis. Based on the observation that the situation appears as a destination $\mathtt{d} = \Phi(addr_1, addr_2 \ldots, addr_n)$ merged by a Φ-function in the SSA form, we propose identifying a shared code segment between the definition site and the use site of the merged destination `d`, which has multiple control flows for each argument address $addr_i$. In other words, we interpret the Φ-function in the merged destination as a "choice operator" and perform inter-procedural analysis by resolving the choice step by step. In a typical situation,

the shared code segment is a subroutine and the arguments of Φ correspond to the callers. Note that the subroutine is called n times.

The evaluation algorithm is described in detail in Section 3.

2.3 Controller

The role of the controller is to manage the interaction between Syman and Alligator.

When a target executable is given, the controller directs Alligator to load it into memory and to prepare the environment for execution. After that, the controller passes the entry address specified in the executable file to Syman for constructing an initial CFG before invoking dynamic execution.

Once an initial CFG is obtained, the controller gets the *current block* to point to the initial basic block of the graph, and suspends Syman and gets Alligator to start step-wise execution from the entry point, confirming that each instruction is contained in the *current block*. After the last instruction of the *current block* is executed, Alligator continues if the next instruction address points at one of the successor blocks of the *current block*, in which case the next block becomes the new *current block*.

The controller suspends Alligator and resumes Syman from the current instruction address to create a *fresh* CFG in the following cases:

- The next instruction address after the execution of *current block* points at none of the successor blocks. This happens when Syman has failed to resolve an indirect jump.
- Alligator reports that the current code page is modified during execution as mentioned in Section 2.1. The *current block* which was read by Syman in the previous phase may differ from actual code which resides in the memory.
- The current instruction address is out of the *current block*. This can be caused by, for example, a run-time exception. The details of the way how a Windows OS handles such run-time exception is omitted due to the space limitation.

After the dynamic analysis terminates (e.g., termination of the emulated process), the controller retrieves a sequence of CFGs created by Syman and puts them together to obtain a single large CFG as the final output of the analysis.

3 Symbolic Value Evaluation Method

In this section, we explain an algorithm to evaluate indeterminate values for extending CFGs. An expression *expr* is called a Φ-expression when *expr* is of the form $\Phi(expr_1, expr_2, \ldots, expr_n)$. Similarly, a statement $var \leftarrow expr$ is called a Φ-statement when *expr* is a Φ-expression. The *location* of a statement s, denoted by $\mathrm{loc}(s)$, is defined as a pair (n, i) where n is a node containing s and i is a non-negative integer such that $i = 0$ when s is a Φ-statement and $i = k$ when s is the k-th non-Φ-statement in n. We write $(n, i)\colon var \leftarrow expr$ when the statement is

located at (n, i). The location of a variable *var*, denoted by $\mathrm{loc}(var)$, is defined by the location of its defining statement if there exists, or $(entry, 0)$ if *var* otherwise. Note that *entry* denotes a designated entry node of a CFG.

We say a location (n, i) is *higher* than (m, j), or equivalently, (m, j) is *lower* than (n, i) when n dominates m $(n \neq m)$, or $i < j$ $(n = m)$. We may write $(n, i) \succ (m, j)$. A special location (n, ∞) is defined for each node n such that $(n, i) \succ (n, \infty)$ for any $i \geq 0$. We say an expression is *valid* if variables appearing in it can be ordered linearly by locations. By nature of control flow graphs in the SSA form, expressions in a CFG are guaranteed to be valid except *Phi*-expressions.

The location of an expression *expr*, denoted by $\mathrm{loc}(expr)$, is defined to be the lowest location of its variables when *expr* is valid, or **undefined** otherwise. For example, every expression appearing on the right-hand side of non-Φ-statements is valid by definition.

3.1 Demand-Driven Evaluation

Suppose that we evaluate a valid expression *expr* in a control flow graph *cfg* of the SSA form. The algorithm assumes a *height limit* to restrict substitution in the evaluation. If it is set higher, evaluation gets more accurate and more time-consuming. A height limit can be any location in *cfg*. We say *expr* is evaluated *up to* location l when the height limit is set to l. Let V be the set of variables appearing in *expr*. If $\mathrm{loc}(expr)$ is higher than the height limit, evaluation terminates and *expr* is returned as the result. Otherwise, a variable v having the lowest location is taken from V and its definition is considered for further evaluation. A new height limit l is set to the location of the second lowest (next to v) variable in $\mathrm{loc}(V)$. If v is a memory state variable M, an outermost subexpression $\mathrm{Ld}(M, addr)$ of *expr* containing M is evaluated up to l using the memory state tracing method explained in the next section. The subexpression is replaced by the evaluation result in *expr*. Otherwise, the defining expression $expr'$ of v found in an assignment $v \leftarrow expr'$ located at $\mathrm{loc}(v)$ is evaluated up to l. If $expr'$ is not a Φ-statement, $expr'$ is further evaluated in a recursive manner and v is replaced by the result in *expr*.

If $expr'$ is a Φ-statement, we locate an assignment $(n, 0)$: $v \leftarrow \Phi(v_0, \ldots, v_k)$. If $(n, 0) \succeq \mathrm{loc}(v_i)$, v_i must have been defined in a node in a loop where n is the loop header, that is, the top node in the loop. We evaluate such v_i up to $(n, 0)$, that is, just for one round of the loop to see if the resulting expression $expr'_i$ contains a recurrence of v. All occurrences of such v are replaced with a placeholder X, which is not treated as a variable in further evaluation. The case where ${expr'}_i$ contains other variables defined at $(n, 0)$ is treated similarly. Since $\mathrm{loc}(expr'_i)$ must be higher than $(n, 0)$, we evaluate ${expr'}_i$ up to l. The other case where v_i is defined at a node out of the loop, we simply evaluate v_i up to l. Suppose all arguments v_i of the Φ-function have been evaluated up to l. Let $\{expr''_i\}$ be the set of resulting expressions. We form $\mu X.\Phi({expr''}_0, \ldots, {expr''}_k)$ as the evaluation result of $\Phi(v_0, \ldots, v_k)$, where μ is the fixed point operator.

Expressions bound by μ are not utilized in further computation except the trivial case where unguarded occurrences of X in a Φ-function are removed, that is, $\mu X.\Phi(X, A, X)$ becomes A. For example, ESP_2 in Figure 4 is defined as $\Phi(\mathrm{ESP}_1, \mathrm{ESP}_5, \mathrm{ESP}_{15})$. Since ESP_5 becomes ESP_2 by evaluation, and ESP_{15} is found to be equal to ESP_1, the result of evaluation of ESP_2 becomes $\mu X.\Phi(\mathrm{ESP}_1, X, \mathrm{ESP}_1) = \mathrm{ESP}_1$. The fixpoint operator is only used for preventing endless recursive computation and no fixed point calculus is attempted at the moment.

For simplicity, we assume throughout the paper that CFGs are *reducible* where all nodes in a loop are dominated by the loop header. For irreducible CFGs, we must carefully treat the loops using *dominator strong components* [19]. It is noted that we keep track of $(n, 0)$ explained above as a loop header during the evaluation along a loop to avoid interminable evaluation in the address comparison in the next section.

Whenever evaluation of *expr* succeeds, the set V of variables appearing in *expr* is updated by taking into consideration variables appearing in replaced subexpressions. If the height limit is initially set higher than any variable appearing in *expr*, locations of variables in V are always lower than the height limit. The above evaluation steps are iterated until V is empty. Note that $\mathrm{loc}(expr) \succeq l$ at this time.

3.2 Memory State Tracing

In this section, we explain ideas for evaluating an expression $\mathrm{Ld}_{sz}(M, addr)$ is to find the most recent memory assignment on the same memory location $addr$.

In this section, we follow an example shown in Figure 4 and explain the case where M's defining assignment $M \leftarrow \mathrm{St}_{sz'}(M', addr', value)$. The other case where M is defined by $M \leftarrow \Phi(M_0, \ldots, M_i)$ is explained in the next section.

Firstly, we compare addresses by evaluating an expression $addr - addr'$ to see if the result d becomes an integer constant such that $0 \leq d < (sz'/8)$. If it is the case, we recursively evaluate *value* up to l. If $d = 0$ and $sz = sz'$, the entire result is returned. Otherwise, required portion of the result is returned, and evaluation continues to obtain the rest. Note that the expression $addr - addr'$ is evaluated as far as possible regardless of l since the height limit is intended for controlling how far we trace the definitions of M and irrelevant for address comparison.

Consider the evaluation of $\mathrm{Ld}(M_{11}, \mathrm{EBP}_1)$ up to $(\mathsf{h}, 0)$ in Figure 4 for example. The evaluation of $\mathrm{EBP}_1 - \mathrm{ESP}_{14}$ should go beyond $(\mathsf{h}, 0)$ until it is confirmed that $\mathrm{ESP}_7 - (\mathrm{ESP}_7 + 12) = -12$ (although this means failure of the comparison because $d < 0$) rather than stopping with the result $\mathrm{EBP}_1 - \mathrm{ESP}_{13} + 4$ at $(\mathsf{h}, 0)$. However, if the comparison occurs in a loop, we limit evaluation up to the loop header since unrestricted evaluation may never terminate.

The address comparison fails when the result d is a non-constant or an integer constant out of range. In this case, we recursively evaluate $\mathrm{Ld}(M', addr)$ instead of $\mathrm{Ld}(M, addr)$. If $\mathrm{loc}(M') \succeq l$, evaluation terminates and $\mathrm{Ld}_{sz}(M', addr)$ is returned. Note that an sz-bit value stored at $addr$ is taken from the virtual

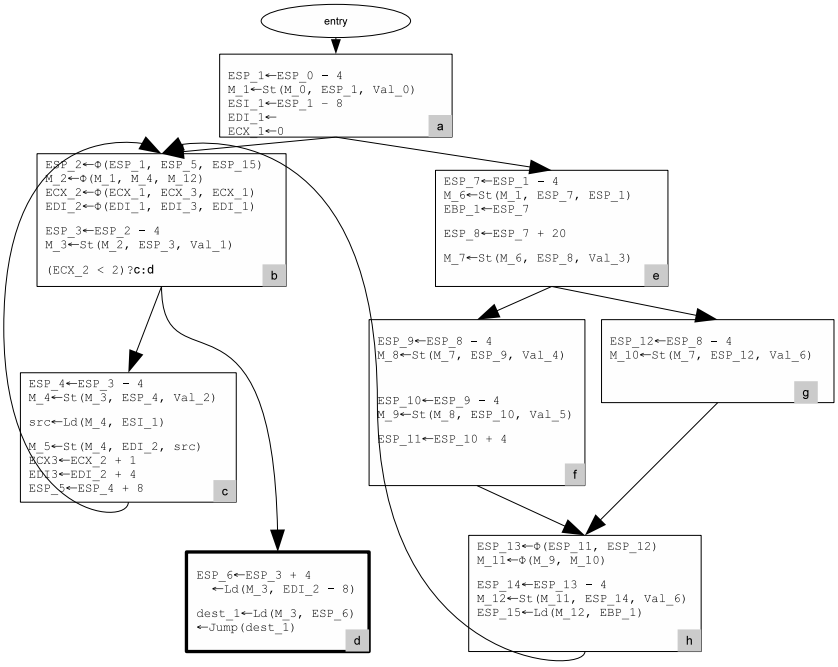


Fig. 4. Memory State Tracing

memory of the system if $l = (entry, 0)$ and in such a case $addr$ is evaluated an integer constant.

Since memory state M is not re-evaluated after comparison failure, there is a chance that the correct value is not obtained. Instead, we may end up with obsolete values. This happens also when one of the addresses in comparison varies in a loop since fixed point calculation is inefficient at the moment. In the current implementation, we may choose to be conservative when stopping address comparison.

3.3 Backtracking over Φ Functions

We discuss the case where M is defined by a Φ-statement in the evaluation of $\mathrm{Ld}(M, addr)$ up to l. For convenience, we write the Φ-statement as $(n, 0)\colon M \leftarrow \Phi(n_0\colon M_0, \ldots, n_k\colon M_k)$, where n_i is a predecessor node of n corresponding to the i-th argument. Since $\mathrm{loc}(addr) \succeq (n, 0)$, if $addr$ has variables defined at $(n, 0)$, they are the lowest among the variables occurring in $addr$ and must have been defined by Φ-statements of the form of $v \leftarrow \Phi(n_0\colon v_0, \ldots, n_k\colon v_k)$. For each n_i, we define $addr_i$ to be an expression obtained by replacing each variable v in $addr$ with v_i. Since $\mathrm{loc}(v_i) \succ (n_i, \infty)$, the derived address $addr_i$ is still valid and $\mathrm{loc}(addr_i) \succ (n_i, \infty)$.

We then evaluate derived expressions $E = \{\mathrm{Ld}(M_0, addr_0), \ldots, \mathrm{Ld}(M_k, addr_k)\}$ by means of backtracking over incoming edges of n. Let $InLoop = \{\mathrm{Ld}(M_i, addr_i)$

$\in E \mid (n,0) \succeq \mathrm{loc}(M_i)\}$ and $\mathit{OutLoop} = E \setminus \mathit{InLoop}$. We first evaluate the expressions in InLoop up to $(n,0)$[1], paying attentions to the confluence of memory state variables. As described in Section 3.2, the loop header $(n,0)$ is regarded as a height limit of address comparison in the evaluation.

If InLoop contains a single expression of the form $\mathrm{Ld}(M_i, \mathit{addr}_i)$, we evaluate it up to $(n,0)$. Otherwise, we choose a pair of expressions of the form $\mathrm{Ld}(M_i, \mathit{addr}_i)$ and $\mathrm{Ld}(M_j, \mathit{addr}_j)$ from InLoop and let M_k be the nearest common variable appearing in the def-use chains of both M_i and M_j. We evaluate these expressions up to $\mathrm{loc}(M_k)$ expecting that they converge. If the results are of the forms $\mathrm{Ld}(M_k, \mathit{addr'}_i)$ and $\mathrm{Ld}(M_k, \mathit{addr'}_j)$ where $\mathit{addr'}_i \neq \mathit{addr'}_j$, we merge them as $\mathrm{Ld}(M_k, \Phi(\mathit{addr'}_i, \mathit{addr'}_j))$. We put back the resulting pair to InLoop instead of the original pair. We iterate the step until InLoop contains at most a single expression of the form $\mathrm{Ld}(M_i, \mathit{addr}_i)$ where $\mathrm{loc}(M_i) \succeq (n,0)$ and evaluate other expressions in InLoop up to $(n,0)$. The set $\mathit{OutLoop}$ is similarly evaluated up to (p,∞). To avoid further recursive evaluation, we replace in InLoop the expressions containing $\mathrm{Ld}(M,...)$ with $\bot$. After that, the expressions in InLoop are evaluated up to (p,∞). Finally, we evaluate $\Phi(\mathit{expr}_0,\ldots,\mathit{expr}_k)$ up to l for $\{\mathit{expr}_i\} = \mathit{InLoop} \cup \mathit{OutLoop}$.

We illustrate the above method on a couple of typical cases occurring in the resolution of destination dest_1 in Figure 4.

Let us first consider the evaluation of $\mathrm{Ld}(M_{10}, \mathrm{EBP}_1)$ up to the location $\mathrm{loc}(M_1)$. In this case, $\mathit{InLoop} = \emptyset$ and $\mathit{OutLoop} = \{\mathrm{Ld}(M_8, \mathrm{EBP}_1), \mathrm{Ld}(M_9, \mathrm{EBP}_1)\}$. We evaluate both expressions in $\mathit{OutLoop}$ up to $\mathrm{loc}(M_6)$. Since EBP_1 does not match any address appearing in the tracing paths, the results coincide as $\mathrm{Ld}(M_6, \mathrm{EBP}_1)$ which is evaluated up to $\mathrm{loc}(M_1)$ (the result is ESP_1).

For the evaluation of $\mathrm{Ld}(M_2, \mathrm{ESP}_2)$ up to $(\mathsf{entry}, 0)$. We have that $\mathit{InLoop} = \{\mathrm{Ld}(M_4, \mathrm{ESP}_5)\}$ and $\mathit{OutLoop} = \{\mathrm{Ld}(M_{11}, \mathrm{ESP}_{15})\}$. However, since both are singletons, we only evaluate their contents. We first evaluate $\mathrm{Ld}(M_4, \mathrm{ESP}_5)$ up to the loop header $(\mathsf{b}, 0)$. This ends up with the recurrence of $\mathrm{Ld}(M_2, \mathrm{ESP}_2)$. We abandon the result since it means that there is (or possibly, for incompleteness of address comparison) no corresponding write operation found in the loop. We next evaluate $\mathrm{Ld}(M_{11}, \mathrm{ESP}_{15})$ up to (a, ∞) to $\mathrm{Ld}(M_1, \mathrm{ESP}_1)$. By merging the results, evaluation proceeds until a single expression $\mathrm{Ld}(M_1, \mathrm{ESP}_1)$. Val_0 is returned as a final result.

4 Implementation Issues

Before presenting case studies, we explain relevant issues on implementation.

4.1 API Blocks

The evaluation method explained in the previous sections is implemented in Syman using Python language. Syman shares virtual memory with Alligator and consults its CPU simulator for the values of registers and memory contents

[1] See the remark on reducibility in Section 3.1.

$$
\begin{aligned}
hModule &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP} + 4) \\
lpProcName &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP} + 8) \\
\mathrm{EAX} &\leftarrow \mathrm{Kernel32GetProcAddress}(hModule, lpProcName) \\
dest &\leftarrow \mathrm{Ld}_{32}(M, \mathrm{ESP}) \\
\mathrm{ESP} &\leftarrow \mathrm{ESP} + 12 \\
&\leftarrow \mathrm{Ret}(dest)
\end{aligned}
$$

Fig. 5. An Example of API Block

prior to static analysis. Alligator is responsible for memory management including loading executables and library file as well as resolving addresses of API functions. Theses API function addresses also concern Syman since going into intricate Win32 library code should be avoided for revealing malware behaviors. In Syman, a special basic block is assigned for each API functions. Figure 5 shows such an API block for GetProcAddress function of Kernel32 library.

It is noted that the `_stdcall` semantics of Win32 platforms is encoded by storing the return valued in EAX register and removing the return address and the arguments pushed onto the stack before the call by increasing the ESP value by the corresponding numbers of bytes. Such numbers may vary for each API function. Syman shares a signature database with Alligator that maintains numbers and types of arguments of functions[2]. In fact, the type information for strings are utilized for evaluating string values by concatenating each evaluated byte data (i.e., character) up to a null character. This typically takes place for LoadLibrary and GetProcAddress to know the names of libraries and functions.

4.2 Caching Evaluation Results

To avoid evaluating the same variable over and over, Syman caches the results of variable evaluation for future use. A cache entry for a variable *var* is a list of pairs (val_i, l_i) where val_i is an evaluation result of `var` up to l_i. When we evaluate *var* up to a location l, we look for the pair (val_i, l_i) having the highest l_i such that $l \succeq l_i$ and evaluate val_i, instead of *var*, up to l. In case $l \neq l_i$, the new result *newval* is cached by appending $(newval, l)$ to the cache entry of *var*. The caching mechanism considerably improves the processing speed of `Syman`.

5 Case Studies

In this section, we present several case studies performed on malware samples gathered in a real environment.

5.1 Defeating Obfuscated Code

We give two examples for obfuscated malware code. The first example, we take a Trojan horse identified as Rustock by the BitDefender scanner[3]. In Rustock, a

[2] The database can be automatically generated.

[3] `http://www.bitdefender.com`

Disassembled Code		SSA Form
401172	MOV SS:[ESP], 4015A9	$M_{106} \leftarrow \mathrm{St}_{32}(M_{105}, \mathrm{ESP}_{208}, 004015A9)$
401179	PUSH 40123B	$\mathrm{ESP}_{209} \leftarrow \mathrm{ESP}_{208} - 4$ $M_{107} \leftarrow \mathrm{St}_{32}(M_{106}, \mathrm{ESP}_{209}, 40123B)$
40117E	RET	$dest_{58} \leftarrow \mathrm{Ld}_{32}(M_{107}, \mathrm{ESP}_{209})$ $\mathrm{ESP}_{210} \leftarrow \mathrm{ESP}_{209} + 4$ $\leftarrow \mathrm{Ret}(dest_{58})$
40123B	RET	$dest_{59} \leftarrow \mathrm{Ld}_{32}(M_{107}, \mathrm{ESP}_{210})$ $\mathrm{ESP}_{211} \leftarrow \mathrm{ESP}_{210} + 4$ $\leftarrow \mathrm{Ret}(dest_{59})$
4015A9	INC ESP	$\mathrm{ESP}_{212} \leftarrow \mathrm{ESP}_{211} - 1$
...	...	...

Fig. 6. Obfuscated Control Flow (Rustok)

```
402F7C MOV DS:1433[EBX]{8}, 'e'
402F85 MOV DS:1435[EBX]{8}, 'p'
402F92 MOV DS:1431[EBX]{8}, 'S'
402F9C MOV DS:1432[EBX]{8}, 'l'
402FA6 MOV DS:1436[EBX]{8}, '\0'
402FB0 MOV DS:1434[EBX]{8}, 'e'
402FD2 LEA EDI, DS:1431[EBX]{8}
402FD8 MOV SS:4[ESP]{32} ,EDI
402FDC CALL DS:1637[EBX]{32}
```

Fig. 7. Hiding API Name String "Sleep" (Troj.Obfus.Gen)

number of RET instructions are inserted to make arbitrary jumps after putting the destination address on the stack top to hide its self-decrypting routine. See the sequence of code excerpted from Rustock in execution order and the translated statements in SSA form in Figure 6. By manipulating the stack beforehand, the two RET instructions at 40117E and 490123B are used to jump to arbitrary addresses instead of returning from procedure calls. The obfuscation appears simple, but is good enough to confuse state-of-the-art disassemblers including IDAPro as they lack data flow analysis capabilities. Statically tracing RET instructions requires serious data flow analysis of evaluating both the stack pointer and the values it points to. Malware creators can take advantage of this to congest analysis processes with automatically generated variants having slightly different obfuscation code.

For Syman, it is an easy task to resolve such obfuscated RET instructions, e.g., $dest_{58}$ and $dest_{59}$ in Figure 6 are resolved as 40123B and 4015A9 respectively. Though Syman succeeds to generate a straight control flow graph for the entire decrypting routine of Rustock having about 103 nodes, its main routine can not be analyzed because of encryption. This is a common limitation of static binary analysis. In such cases, Alligator helps Syman to proceed as explained before.

Secondly, we take an example of a malware sample named Troj.Obfus.Gen by BitDefender scanner of approximately 17.5KB size. It is an unencrypted Trojan horse that opens Internet connection using WININET.DLL library and serves as a good example of obfuscated code. As its name suggests, Troj.Obfus.Gen tries to hide the name of the API functions it calls or the library it dynamically load by decomposing the name string into groups of characters, and distributing them through various registers and memory locations. See the code excerpt from Troj.Obfus.Gen in Figure 7, in which the irrelevant code are omitted. In this example, an API name "Sleep" (including the last null character '\0'), given as the second argument for a GetProcAddress library call (corresponding the last CALL DS:1637[EBX]{32}), is hidden in a sequence of MOV instructions. Although an advanced disassembler including IDAPro can suggest API function calls for memory indirect CALLs for which the target address is trivial, an obfuscated case as described above can not be handled and no further information is obtained in analysis.

Syman has successfully analyzed more than 89% of the code in the size of the code section declared in the executable file. Considering the code alignment and padding bytes in the gap, it is fair to say that Syman's analysis is complete for this example.

There is only a single indirect destination which Syman failed to resolve in Troj.Obfus.Gen. It is a `CALL` instruction direct to an address at midpoint of a heap memory allocated by VirtualAlloc API function. The program tries to download a executable file from the Internet using WININET.DLL library API, store it into the heap, search an entry point in the downloaded executable, and then jump to the address obtained by the search computation with loops. Such address is obviously out of scope for static analysis and should be resolved by a dynamic analyser. Alligator is capable of doing thie.

5.2 Automated Analysis of Self-extracting Code

We experimented on a number of variants of online game worms identified as `Trojan.PWS.OnlineGames.*` collected in the year of 2009. These worms are encrypted by execution packers. We demonstrate how our method treats such samples by alternating Syman and Alligator.

After Alligator loads a sample into memory, Syman constructs an initial CFG covering the decryption routine to pass control to Alligator which in turn executes the routine. When Alligator finishes the decryption, it jumps to the decrypted code block on a page modified during execution. Alligator judges that it has encountered self-modification of code and let Syman analyze the code from the newly found block.

For many samples, the first thing to do after decryption is to retrieve the entry addresses of API functions and to store them in a global function table in order. Those API functions are later called directly by consulting the addresses in the table. Since the API functions stored in the table can not be statically determined, Syman creates the second CFG until it meets the first API call through the table. When Alligator reaches the `CALL` instruction, the API function to be called is now determined. Syman resumes analysis from the API function block. In this turn, the API function on the table are accessible by Syman. The main body of the sample is obtained as the third CFG, in which the CreateThread function may be called. In such cases, Alligator invokes Syman to analyze the thread code before emulating thread creation.

As demonstrated above, our method can automatically analyze self-extracting malware code even in multiple encryption layers.

5.3 Processing Speed

In this section, several performance benchmark data is presented to show the current status of the tools. These data are collected on a PC with Quad-Core Intel Xeon CPU (3.0GHz) with 16GB RAM running under Linux kernel 2.6.24.

- Rustock: 79secs for analyzing the decrypting routine.
- Troj.Obfus.Gen: 1588secs for entire analysis
- online game worms: approximately from 20mins to 50 hours.

Since `Syman` is implemented in Python language, the processing speed is not as good as we wish. We are working to improve the efficiency of evaluation.

6 Related Work

The limitation of single-path execution in dynamic analysis has been tackled by Moser and others [13]. The method tracks "tainted" values such as run-time system information and data obtained via network to identify conditional branches which depend upon them. When one of those branches is met, the execution context is saved once and restored after termination of a single execution to explore the other path by adjusting the tainted variables on which the branch depends. The method is reported to allow multi-path exploration in a dynamic fashion although the cost of saving/restoring the execution contexts may become costly as the number of branches to be checked increases. It is called the "path explosion" problem. Manipulating conditional branches without knowledge of global control flows also runs a risk of coming to a deadlock caused by inconsistency and explosive growth of the number of paths to be explored. A similar approach is reported by Brumley and others [5] that performs logical symbolic execution over multiple paths rather than context switching. The method seem to suffer from the same path explosion problem. These methods can be rendered ineffective by hiding trigger conditions as reported by Sharif and others [17], using a source code transformation technique introducing hash functions. The method proposed in this paper, on the other hand, does not depend on trigger conditions but on the destination addresses of branches at the machine instruction level. It will not be affected by trigger condition hiding alone.

A rather different approach is taken by Milani and others [7] that abstracts malicious code fragments detected by dynamic analysis with a static analysis method similar to slicing. By comparing obtained abstract models against other samples, they successfully identified a number of common malicious behaviors that would have escaped usual dynamic analysis. However, the static analysis method they used is rather simple and no extensive date flow analysis is performed. The method is still vulnerable to advanced packing techniques and code obfuscation, which are common in modern malware.

Resolution of target addresses of indirect jumps is studied in various contexts. Cifuentes and Emmerik proposed a solution for recovering jump tables from binary code [6], in their efforts toward decompilation. A jump table is an array of addresses generated by compilers for transferring to different program locations with a single indirect jump. The approach examines disassembled instructions backward from the indirect jump to find a sequence of instructions to fetch the target address which typically involves adding an index number to the base address of the table. The method requires the target executable to be compiled

from C-like language by a specific compiler as it relies on known code patterns. Our method is capable of tracking control flows over such switching structures generated by normal compilers.

The *value set analysis (VSA)* [2] is a technique developed by Balakrishnan and Reps for static analysis of binary code. It is based on abstract interpretations using approximation over sets of possible values stored in variable-like memory locations, which they call *a-locs*. The method assumes that the target binary is generated by a standard compiler and tries to identify memory locations corresponding to variables appearing in the original source code by heuristic analysis. The method depends on a commercial disassembler called IDAPro[4] for a-loc discovery and initial CFG construction, after which fix point calculation is performed to estimate possible value ranges of a-locs via over-approximation.

While the method focuses on compiler generated executables, there are many malware samples written in low-level assembly languages and not amenable to heuristic discovery of variables. This is why we developed an on-demand/backward value evaluation method since the instances of registers and memory locations that may hold indeterminate addresses can not be enumerated before analysis for such malware code. By its nature, the VSA is unable to analyze malware code when it is not compiler generated and is obfuscated well enough to cheat IDAPro, which is possible, for example, by inserting fake `RET` instructions after manipulating the stack.

The evaluation step of our method employs a constant propagation algorithm in SSA form called the sparse simple constant propagation (SSC) algorithm [15,16]. It traces the def-use chains in SSA form to replace a variable with a constant value when the definition is constant as described in Section 3.1.

Binary code analysis has become a major topic in the area of computer security. The BitBlaze Binary Analysis Platform Project at UC Berkeley is one such prolific example[5]. It advocates coordinated use of static and dynamic methods for binary code analysis. In its static analysis component called Vine [4], SSA form is used for efficient calculation and representation of weakest preconditions for which basic forward propagation is performed to simplify internal expressions in the form of symbolic execution. The value set analysis explained above is employed for resolving indirect jumps/calls. Unresolved memory access will be made concrete with a help from dynamic analysis, that is, real code execution in virtual environments. As has been pointed out, Syman is more suited for malware analysis than the VSA. Vine will benefit from the capabilities of Syman in this regard.

7 Conclusion and Future Work

In this paper, we have presented a malware analysis method that uses static analysis techniques to resolve indirect jumps for further exploration of control flows. The method is free of artificial limitations on the target binaries and is

[4] `http://www.hex-rays.com/idapro/`
[5] `http://bitblaze.cs.berkeley.edu/`

able to extract control flows not examined by dynamic analysis. Experiments show that entire CFG can be generated for malware samples armed with packers and obfuscation. We have developed a static analysis tool called Syman that implements an on-demand symbolic evaluation algorithm on the static single assignment form of binary code, where values passing through memory are traced by locating matching read/write operations. A previously developed dynamic analysis tool (e.g., an emulator) called Alligator is used to overcome the limitations of the static analysis.

Generated CFGs can be used for many purposes including detection of unknown malware, multi-path analysis, and identification of dormant malicious behaviors. For further analysis on CFGs, we are planning to apply a tree-difference algorithm [10] on dominator trees derived from CFGs for measuring distance/similarity among them. This will help us to establish an automated method for semantic classification/identification of malware.

We plan to enhance abilities of Syman by introducing extended versions of the SSA form such as gated single assignment (GSA) [3,20] or the static single information (SSI) [1] form to better handle loops and conditional branches. Alligator also needs improvements, especially its emulation capabilities against various anti-debugger/emulator techniques. Since Alligator works as a self-contained software tool, it is not difficult to modify and improve its functions as opposed to full-fledged emulators requiring working Windows OS instances. We are currently working to incorporate Syman into a hardware based virtualization tool such as Ether [9] to build a fully automated platform for malware analysis.

Acknowledgements

The research described in this paper has been supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) under management of the Ministry of Internal Affairs and Communications (MIC) of Japan.

References

1. Ananian, C.S.: The static single information form. Tech. Rep. MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology (September 1999), `http://www.lcs.mit.edu/specpub.php?id=1340`
2. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
3. Ballance, R.A., Maccabe, A.B., Ottenstein, K.J.: The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pp. 257–271 (1990)
4. Brumley, D.: Analysis and Defense of Vulnerabilities in Binary Code. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (2008)

5. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Lee, W., et al. (eds.) Botnet Analysis and Defense (2007)
6. Cifuentes, C., Van Emmerik, M.: Recovery of jump table case statements from binary code. Science of Computer Programming 40(2-3), 171–188 (2001)
7. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: IEEE Symposium on Security and Privacy, pp. 61–76. IEEE Computer Society, Los Alamitos (2010)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
9. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: CCS 2008: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 51–62. ACM, New York (2008)
10. Hashimoto, M., Mori, A.: Diff/TS: A tool for fine-grained structural change analysis. In: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE (2008)
11. Mori, A.: Detecting unknown computer viruses – a new approach. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 226–241. Springer, Heidelberg (2004)
12. Mori, A., Izumida, T., Sawada, T., Inoue, T.: A tool for analyzing and detecting malicious mobile code. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 831–834 (2006)
13. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: IEEE Symposium on Security and Privacy, SP 2007, pp. 231–245 (2007)
14. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: 23rd Annual Computer Security Applications Conference, ACSAC (2007)
15. Reif, J.H., Lewis, H.R.: Symbolic evaluation and the global value graph. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 104–118 (1977), http://doi.acm.org/10.1145/512950.512961
16. Reif, J.H., Lewis, H.R.: Efficent symbolic analysis of programs. Journal of Computer and System Sciences 32(3), 280–313 (1986), http://dx.doi.org/10.1016/0022-00008690031-0
17. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA (2008)
18. Stoltz, E., Wolfe, M., Gerlek, M.P.: Constant propagation: A fresh, demand-driven look. In: Symposium on Applied Computing. ACM SIGAPP, pp. 400–404 (1994)
19. Tarjan, R.E.: Fast algorithms for solving path problems. Journal of the ACM 28, 594–614 (1981)
20. Tu, P., Padua, D.: Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In: Proc. 9th International Conference on Supercomputing (ICS 1995), pp. 414–423. ACM Press, Barcelona (1995)