Semantic Malware Detection

Technical Report TR-10-03, 16 February 2010

Khalid Alzarouni David Clark Laurence Tratt

E-mail: khalid.alzarouni@kcl.ac.uk, david.j.clark@kcl.ac.uk, laurie@tratt.net

Abstract

Polymorphic and metamorphic malware use code obfuscation techniques to construct new variants which preserve the semantics of the original but change the code syntax, evading current compiled code based detection methods. Dynamic slicing is a technique that, given a variable of interest within a program, isolates a relevant subset of executed program code that influences that variable. Using dynamic slicing to condition semantic traces identifies 'core' behaviours that, as part of an overall semantics based approach, has the potential to play a significant rôle in detecting difficult malware variants. We preface this by a discussion of the motivation and the contextual role for this form of slicing in semantics based matching. A brief outline of the semantic trace mapping algorithm is presented with an example. We complete the report with presentation of our test data generation technique using backward domain reduction with some examples as a stand-alone step in the process of genearting data inputs for producing unique semantic program traces.

1 Overview of the detection process

We begin this document by presenting a brief overview of how we plan to use semantics to detect malware variants. This section is the work of David Clark and Laurence Tratt. The remainder of the document is the work of Khalid Alzarouni and David Clark with occasional advice from Laurence Tratt.

1.1 The approach and some applications

We present an outline solution to a specific problem: the ability of polymorphic and metamorphic malware to use semantic equivalence tables together with a rewriting engine to alter their signature each time they are copied. The key idea is to use a finite set of finite semantic traces as a semantic version of a "signature" for a known malware. Once the signature is extracted a semantic simulator can be used to attempt to match behaviours.

With the challenge of real-time detection of variants in mind we present the scheme in two phases. The initial, preparatory phase is when the semantic signature is constructed via analysis of the malware. The second, detection phase attempts to match semantic signatures with a candidate program. For the purposes of this outline we do not treat the question of infection but this is a straightforward extra step using abstraction and trace matching techniques.

The outline given in what follows is not the only possible way to instantiate our framework. Our proposal offers the following advantages over existing detection methods:

- This approach can be used in much the same way that signatures are currently used to detect malware but clearly can shorten and decentralise the process of recognising new malware and distributing new signatures.
- It could also speed up 'central' detection of malware variants by anti-virus software providers, for example.
- A possibility is the use of the approach in real-time detection of malware variants.

1.2 Preparatory phase: constructing the semantic signature

We assume that we have a known malware program, identified from its syntactic signature using standard methods. The known malware may not be "vanilla", i.e. may not be the original program before metamorphosis or polymorphosis, but may be the result of several transformations. What is important is that it retains the same behaviour as the original, "vanilla" program. Note that in this initial phase we aim to detect variants of known programs rather than different programs with different behaviours. This latter problem could be tackled using a variation on our approach. This would require a research program to develop 'positive' security guidelines which unknown malware with unknown behaviours would violate. The semantic signature we construct is not a complete semantics for the malware. Technically it is an abstraction of a slice of part of the semantics of the malware. Since it is only part of the semantics there can never be iron-clad guarantees that it will always detect a variant or that what it detects is always a variant. However there are some things we can guarantee, for example that any manipulations we perform on the semantics in the course of developing the signature are "safe" in the sense that they do not introduce anything that was not there to begin with. In addition, the selection of the part of the semantics used in the signature is done with a guarantee of a certain type of limited coverage with respect to all the possible behaviours of the malware. Finally, note that each step in the preparatory phase may be allowed to be performed with some human intervention as there is no real-time consideration.

- *Reverse engineer the compiled code*. The first step is to distinguish code and non-code and turn the binary code into a simple assembly language or equivalent.
- *Generate the Control Flow Graph*. This is the first abstraction on the malware.
- *Analyse the CFG*. Analysis of the CFG can be partly or possibly entirely automated. The aim of the analysis is to extract a set of test inputs that produce a finite set of finite execution traces that cover all reachable executable statements in the malware. This does not necessarily cover all possible behaviours of the malware. Call the test set T.
- *Generate the semantic traces*. Run the malware on the inputs in the test set, T, using a semantic simulator for the assembly language. Each input produces a corresponding semantic trace which contains information about the evolution of both the code and the state of the machine during the execution run.

- *Slice the semantic traces*. Slicing is a syntax based algorithm which produces a smaller program that has the same semantic effect with respect to a slicing criterion. Here we slice on the value of all variables at the end of the trace. This in general produces a smaller trace and can remove some effects of obfuscation. Even a small reduction in the trace length can improve the speed of the matching algorithm performed later in the detection step.
- *Abstract the sliced semantic traces*. Finally, abstractions are applied to each sliced trace. The basic one is to remove the syntax of the commands from the traces, retaining the information about the evolution of the state, but there are others which can deal with infection, variable renaming, and other obfuscations. Call the set of abstracted, sliced traces $\mathcal{TR}$.
- *Semantic Signature*. Each sliced trace corresponds to an input from T. The semantic signature, $\mathcal{P}$, is then the set of pairs consisting of each input, with its resulting, sliced trace.

$$\mathcal{P} = \{(t_i, \tau_i) \mid t_i \in T, \tau_i \in \mathcal{TR}, \text{ where } \tau_i \text{ is generated by } t_i \text{ using } M\}$$

1.3 Detection phase: using the semantic signature

- *Reverse engineer the compiled code*. Here we automatically generate the assembler code for a candidate program, C.
- *Generate semantic traces*. The tests developed in the previous phase are applied to the candidate using the semantic simulator. For each test $t_i \in T$ a semantic trace in $\mathcal{TR}'$ is generated.
- *Abstract the semantic traces*. This step corresponds to the abstraction of the sliced traces in the Preparatory phase and and can be automated.
- *Candidate's semantic signature*. After abstraction we have the candidate's semantic signature, $\mathcal{C}$, again a set of pairs.

$$\mathcal{C} = \{(t_i, \tau_i') \mid t_i \in T, \tau_i' \in \mathcal{TR}', \text{ where } \tau_i' \text{ is generated by } t_i \text{ using } C\}$$

- *Match traces*. For each t_i the corresponding signature traces for M and C are compared. The algorithm used is a form of sub trace inclusion match, in which for each t_i the algorithm checks whether the sequence of nodes in τ_i occurs in τ_i', possibly with interpolations of extraneous nodes.

In what follows we flesh out this overview.

2 Introductory remarks

Malware writing and malware detection is big business. It is a combative and fast evolving part of IT and computer science. From the detection point of view the gold standard for some time has been the libraries of 'signatures' which must be kept complete and as up to date as possible. These libraries allow recognition of stored, known malware without false positive identifications. One weakness of the libraries approach is the time gap between identification and signature dissemination. As long as this gap is short this weakness is acceptable. In recent years the development and increasing proliferation of self-modifying metamorphic and polymorphic malware has dramatically sped up the production of malware variants. The former use a table of semantic equivalences to preserve semantics while altering the syntax of the machine code so that after execution the stored executable has a different signature. This report takes a single but significant step in the direction of detecting such variants on a known program.

The primary contributions of this report are threefold. It identifies a suitable assembly language which is sufficiently expressive to be able to become a representative of widely used assembly languages such as FASM [1] and NASM [2], and has a sufficiently well defined semantics so as to be amenable to semantics based analysis. It defines a notion of slicing for the traces that occur in the semantics and provides a slicing algorithm. Finally, it proves that this algorithm is correct. This report also provides a discussion of the strengths and weaknesses of the slicing semantic traces approach with respect to finding malware variants, although no formal guarantees are provided at this stage. Last but not least, the report discusses briefly the algorithms of mapping semantic traces with respect to a set of obfuscation techniques which are incorporated with the trace slicing algorithm. To fully appreciate this contribution, it is necessary to understand the context in which we propose to apply the slicing algorithm.

Our overall approach is based on the semantic simulation of the execution of a suspected malware program. We assume that this program is a *conservative*, simple obfuscation of another. It is conservative because there has been no variable renaming (although fresh variables may have been introduced) during the transformation and it is simple because it is not an infection (or contained within) another program. Even with these assumptions there are some considerable theoretical difficulties in using semantic traces directly. Determining whether one program is semantically equivalent to another is not in general decidable, or even partially decidable. Consequently, it is not possible in the general case to provide safety guarantees for semantics based detection in the traditional correct program analysis style. Either detection is general and partial, but statistically significant (low false positive rate), or guarantees are absolute but detection is specific to a limited, fixed set of transformations [19]. We have developed this slicing algorithm as part of an overall approach which aims to be general and partial.

Specifically, we intend that there is an initial analysis phase for the known malware. In this phase it is reverse engineered to an AAPL program and its control flow graph (CFG) is extracted. On the basis of the CFG a set of test inputs, $\mathcal{I}$, are derived which guarantee a coverage property, manifested as a finite set of finite traces, with respect to the CFG. We call this set of finite traces the *approximation semantics* of the malware. The traces in this approximation semantics are then conditioned by dynamic backward slicing using all variable values at the end of the trace. This conditioning produces smaller traces, closer to traces of the vanilla (or unobfuscated) malware, reducing the complexity of the abstract trace matching phase. Further abstraction removes command syntax and retains execution contexts (i.e., program environment and memory sequences). Then the detection algorithm uses each semantic trace slice pairs (i.e. trace slices of a known malware program and an obfuscated variant) as graphs to identify multiple potential mappings between the pair of traces. The slicing algorithm improves the detection of malware variants in two ways. First, the algorithm detects code obfuscating techniques and removes the effects of code obfuscations. Second, it computes a set of correct semantic trace slices for the malware matching algorithm (detector) to inspect against semantic trace slices of a known malware program. Thus, the slicing approach helps the malware detector in producing fast and accurate detection results.

Moreover, we present a method for mapping semantic traces of program executions of two program variants. The mappings generated can be the key to detect and determine if one program is a variant of another program. This can be useful when obfuscation techniques are deployed during the generation of new malware variants. Unlike some static analysis approaches for mapping and detecting program variants, our method is implemented at the level of executable binaries of the two program variants and does not require access to the programs source code. In particular, the approach refines the set of mappings by comparing the execution contexts (i.e. *semantics*) associated with the mapped trace slices.

The remainder of this report is structured as follows. Section 3 explains the syntax the semantics of our programming language, AAPL. Section 4 presents the semantic trace slicing algorithm and its correctness proof. Section 5 highlights the strengths and limitations of the algorithm. Section 6 describes the related research work in the area of dynamic program slicing and slicing binary executables. Section 7 briefly discusses the actual approach of mapping individual semantic traces and presents its algorithms. Section 8 outlines our approach in finding a set of test data inputs via backward domain analysis technique. Section 9 concludes the report.

3 Programming Language

In this section we introduce our simple abstract assembly programming language (AAPL) which is used by our dynamic slicing algorithm and for reasoning about code obfuscating transformations in malware program variants. Our main objective is to have an indicative intermediate representation of assembly programs that aid in supporting various program analysis approaches such as generating CFGs, PDGs, etc.; moreover, this approach allows us to investigate semantic properties of code independently of the target architecture. This enables its use to employ source analysis techniques on low-level code.

3.1 Syntax

Programs written in AAPL consist of a sequence of statements. Every program statement contains a command C and, optionally, a label L. We define program registers to be a finite set of assembly registers which represent a small fixed set of word-sized containers during program execution. We define PC as the program counter register to hold the memory address of the next command to be executed and SP as the stack pointer register which points to a region of memory. Our programming language semantics are similar to those presented in [19], except that our language treats memory addresses as unsigned integer numbers $\mathbb{Z}_\perp$ and assumes they hold either integer values or commands.

$R ::= \{PC,\ SP,\ R0,\ R1,\ ...,\ Rn\}$
$E ::= n \mid L \mid R \mid *E \mid E_1\ \texttt{op}\ E_2 \quad (\texttt{op} \in \{+, -, *, /, ...\})$
$B ::= true \mid false \mid E_1 < E_2 \mid \neg B_1 \mid B_1\ \texttt{\&\&}\ B_2$
$A ::= R := E \mid \texttt{SKIP} \mid \texttt{JMP}\ E \mid *R := E \mid \texttt{CALL}\ E \mid \texttt{RTN}$
$C ::= C_A := A$
$\quad \mid C_B := B\ \texttt{JMP}\ E$
$P ::= \Sigma(\mathbf{C})$

(a)

$\mathbb{B} = \{true, false\}$	(truth values)
$n \in \mathbb{Z}$	(unsigned integers)
$\rho \in \mathcal{E} = \mathbf{R} \rightarrow \mathbb{Z}_\perp$	(environments)
$m \in \mathcal{M} = \mathbb{Z} \rightarrow \mathbb{Z}_\perp \cup \mathbf{C}$	(memory)
$\xi \in \mathcal{X} = \mathcal{E} \times \mathcal{M}$	(execution contexts)
$S = \mathbf{C} \times \mathcal{X}$	(program states)

(b)

Figure 1. Instructions Syntax and Value Domain

Figure 1 describes the programming syntax of AAPL. A program P is a sequence of commands $\Sigma(\mathbf{C})$. There are two types of commands in AAPL, actions and conditional jump commands. An action command C_A may perform the following: evaluating an expression to a register ($R := E$), loading the result of an expression into a memory location pointed to by a register, performing *SKIP* (i.e. nop) operation. An unconditional jump command may perform jumps based on an expression value, a call by expression value and a return to a memory location specified by the stack pointer SP. A conditional jump command C_B performs a jump to a location specified by the value of expression E when the Boolean expression B evaluates to $true$ (e.g. $\hat{\mathbf{B}}[\![B]\!] = true$).

In Figure 1b, we let ρ describe the *environment* of program registers, including the program counter, during program execution. An environment $\rho \in \mathcal{E}$ maps a register to its content value, i.e. $\rho : \mathbf{R} \rightarrow \mathbb{Z}_\perp$. Moreover, the *memory* in the language describes the actual contents of program registers and locations represented by arithmetic expressions.

3.2 Semantics

The semantics of the programing language is presented in Figure 3. The semantics of actions describes how the memory and the environment pair (ρ', m') of the next command to be executed in the program is evaluated. The execution of program $P = \Sigma(\mathbf{C})$ starts by executing the initial command of P that is specified by the program counter PC. PC always points to the memory location of the first command in the program. That is, a sequence of program commands stored in the memory are reachable through execution at runtime via memory locations pointed to by PC. The memory location values are computed during program execution and assigned to PC. Thus, PC should hold a valid memory address. For instance, when executing a call command, the location of the next command in the program is stored in the stack memory indexed by SP. Also, in the semantics of return command (RTN), the program counter retrieves the location of the next command to be executed from the stack.

The behaviour (i.e, the set of traces) of a program during the execution is described by the set of *execution contexts* $\mathcal{X}$, where $\mathcal{X} = \mathcal{E} \times \mathcal{M}$ is a pair of the environment and memory of the program being executed [19]. A program execution *state* $s \in S$ is a pair of command and execution context, (C, ξ). The set of program execution states, denoted by $S = (\mathbf{C} \times \mathcal{X})$, describes both the program command and the execution context of the program in each state. The transition function $\hat{\mathbf{C}} : S \overset{PC}{\rightarrow} \Sigma(S)$ specifies transition relation between states by determining the memory content pointed to by PC and evaluating the next command to be executed. That is, for a given state s $\hat{\mathbf{C}}(s)$ provides the next program state s' via evaluating s. For instance, for the unconditional jump command, $\texttt{JMP}\ E$, the arithmetic expression E in the current command must be evaluated and the result is assigned to the program counter PC which represents the location of next program command (i.e. $C' = m(\rho(PC))$). Figure 2 shows a fragment of a malware routine written in AAPL and its single execution trace, which represents the the program environment and memory evolution (the notation of the execution trace T_{tc} is explained in the following section).

4 The Semantics Trace Slicing Algorithm

We consider T^* to be the set of finite sequences of program execution states. A program execution trace $T_{tc} \in T^*$ consists of a sequence of program states $< s_1, ..., s_n >$ of length $|T_{tc}| \geq 0$ that has actually been produced by executing the program with initial state s_0 and a test input tc: $s_i \in \hat{\mathbf{C}}(s_{i-1})$ for all $i, 1 \leq i \leq n$.

P :

```
1          R0:=n
2          R1:=m
3 Loop:    (R0 >= 3) JMP Exit
4          *R1:=*R0+4
5          R0:=R0+1
6          R1:=R1+1
7          JMP Loop
8 Exit:    JMP ...
```

(a)

T_{tc} :

s_1	R0:=n, $(\rho_{s_1}(\texttt{R0}\mapsto 1,\texttt{R1}\mapsto 0), m_{s_1}(\texttt{R0},\texttt{R1}))$
s_2	R1:=m, $(\rho_{s_2}(\texttt{R0},\texttt{R1}\mapsto 2), m_{s_2}(\texttt{R0},\texttt{R1}))$
s_3 Loop:	(R0 >= 3) JMP Exit, $(\rho_{s_3}(\texttt{R0},\texttt{R1}), m_{s_3}(\texttt{R0},\texttt{R1}))$
s_4	*R1:=*R0+4, $(\rho_{s_4}(\texttt{R0},\texttt{R1}), m_{s_4}(\texttt{R0},\texttt{R1}\mapsto (m_{3^3}(\texttt{R0})+4)))$
s_5	R0:=R0+1, $(\rho_{s_5}(\texttt{R0}\mapsto 2,\texttt{R1}), m_{s_5}(\texttt{R0},\texttt{R1}))$
s_6	R1:=R1+1, $(\rho_{s_6}(\texttt{R0},\texttt{R1}\mapsto 3), m_{s_6}(\texttt{R0},\texttt{R1}))$
s_7	JMP Loop, $(\rho_{s_7}(\texttt{R0},\texttt{R1}), m_{s_7}(\texttt{R0},\texttt{R1}))$
s_8 Loop:	(R0 >= 3) JMP Exit, $(\rho_{s_8}(\texttt{R0},\texttt{R1}), m_{s_8}(\texttt{R0},\texttt{R1}))$
s_9	*R1:=*R0+4, $(\rho_{s_9}(\texttt{R0},\texttt{R1}), m_{s_9}(\texttt{R0},\texttt{R1}\mapsto (m_{s_8}(\texttt{R0})+4)))$
s_{10}	R0:=R0+1, $(\rho_{s_{10}}(\texttt{R0}\mapsto 3,\texttt{R1}), m_{s_{10}}(\texttt{R0},\texttt{R1}))$
s_{11}	R1:=R1+1, $(\rho_{s_{11}}(\texttt{R0},\texttt{R1}\mapsto 4), m_{s_{11}}(\texttt{R0},\texttt{R1}))$
s_{12}	JMP Loop, $(\rho_{s_{12}}(\texttt{R0},\texttt{R1}), m_{s_{12}}(\texttt{R0},\texttt{R1}))$
s_{13} Loop:	(R0 >= 3) JMP Exit, $(\rho_{s_{13}}(\texttt{R0},\texttt{R1}), m_{s_{13}}(\texttt{R0},\texttt{R1}))$
s_{14} Exit:	JMP ...

(b)

Figure 2. A sample program in AAPL and its execution trace. (a) a sample program; (b) an execution trace on input: $n = 1, m = 2$

The execution trace T_{tc} of a program captures the complete runtime information of the program's execution, which can later be used by our slicing algorithm. The information that the trace holds consists of both the command (*the syntax*) trace and execution context reference (*the semantics*) trace. For example, $T_{tc} =< s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14} >$ is the program execution trace when the program in Figure 2 is executed on the input data $n = 1, m = 2$. Notationally, each program state in an execution trace is subscripted with its position. We let POS_T to denote the set of positions of program execution states S in a program execution trace T. Also, in order to map a particular execution position to the execution state in T, we define the auxiliary function $State : POS_T \rightarrow S$.

Unlike some traditional slicing algorithms proposed in the literature [3, 13], where the control flow graph of the program is *statically* analysed and the full dynamic program dependence graph (DPDG) is constructed in order to perform the slice, the algorithm that we present, *Semantic Trace Slicing* (STS), does not require the computation of either control dependencies or the program dependence graph. Instead, the STS algorithm involves the following: *on-the-fly* computation of data dependence edges from the trace, constructing *dynamic data dependency* graph and performing the slice for a given slicing criterion. Therefore, the trace slice, which is computed from the program execution trace, is the transitive closure of data dependencies in the *DDDG* relevant to the trace slicing criterion. The following three subsections present definitions, overview and a description of the STS algorithm.

4.1 Definitions

We present a few definitions that are included in the STS algorithm. In these definitions, and throughout the rest of the report, we use the term *state* nodes to denote program execution states in an execution trace. Also, AAPL uses registers, R (i.e, the environment $\rho(R)$) and direct memory locations (i.e, addressing memory locations with an immediate offset, a register, or a register with an offset) in order to perform data manipulations during program execution, such as retrieving and storing data from memory.

We use the term *data manipulator* to denote registers and memory locations that are used to process the program data.

Definition 1 (data manipulator (DM))**.** *In AAPL, a* data manipulator *is a program register or memory location used to perform data definition and manipulation operations. The value of a data manipulator is described as either the environment value, $\rho(DM)$ in the case of a register or the memory value $m(DM)$ in the case of a memory location.*

During program execution DM can be defined or used at any point via a state node (e.g. assignment or memory update operations). In order to capture data dependency information in an execution trace, the following definitions are introduced.

Semantics of Arithmetic Expressions:

$\hat{\mathbf{E}} : \mathbf{E} \times \mathcal{X} \rightarrow \mathbb{Z}_{\perp}$

$\hat{\mathbf{E}}[\![n]\!]\xi = n$

$\hat{\mathbf{E}}[\![L]\!]\xi = n$

$\hat{\mathbf{E}}[\![R]\!]\xi = \rho(R)$

$\hat{\mathbf{E}}[\![*E]\!]\xi =$ if $(\exists n\ \hat{\mathbf{E}}[\![E]\!]\xi\ \in n)$ then $m(n)$; else $\perp$

$\hat{\mathbf{E}}[\![E_1\ \texttt{op}\ E_2]\!]\xi =$ if $(\hat{\mathbf{E}}[\![E_1]\!]\xi \in \mathbb{Z}$ and $\hat{\mathbf{E}}[\![E_2]\!] \in \mathbb{Z})$ then $\hat{\mathbf{E}}[\![E_1]\!]\xi\ \texttt{op}\ \hat{\mathbf{E}}[\![E_2]\!]\xi$; else $\perp$

Semantics of Actions:

$\hat{\mathbf{A}} : \mathbf{A} \times \mathcal{X} \rightarrow \mathcal{X}$

$\hat{\mathbf{A}}[\![\texttt{SKIP}]\!]\xi = \xi$ where $\xi = (\rho, m)$

$\hat{\mathbf{A}}[\![R := E]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(R \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![*R := E]\!]\xi = (\rho, m')$ where $\xi = (\rho, m)$ and $m' = m(\rho(R) \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![\texttt{JMP E}]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![\texttt{CALL E}]\!]\xi = (\rho', m')$ where $\xi = (\rho, m),\ \rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi, SP \mapsto SP - 1)$ and $m' = m(\rho(SP - 1) \mapsto \rho(PC + 1))$

$\hat{\mathbf{A}}[\![\texttt{RTN}]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(PC \mapsto m(\rho(SP)), SP \mapsto SP + 1)$

Semantics of Commands:

$\hat{\mathbf{C}} : S \stackrel{PC}{\rightarrow} \Sigma(S)$ (determines transition relation between states via PC)

$\hat{\mathbf{C}}[\![C_A]\!]\xi = (\xi', C')$ where $\xi = (\rho, m),\ \xi' = \hat{\mathbf{A}}[\![A]\!]\xi$ and $C' = \begin{cases} m(\rho(PC)) & \text{if } A := \texttt{JMP} \cup \texttt{CALL} \cup \texttt{RTN} \\ m(\rho(PC + 1)) & \text{otherwise} \end{cases}$

$\hat{\mathbf{C}}[\![C_B]\!]\xi = (\xi', C')$ where $\xi = (\rho, m),$ and

$$(\xi', C') = \begin{cases} \xi' = (\rho', m), \rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi), C' = m(\rho(\hat{\mathbf{E}}[\![E]\!]\xi)) & \text{if } \hat{\mathbf{B}}[\![B]\!]\xi = \text{true} \\ \xi' = \xi, C' = m(\rho(PC + 1)) & \text{otherwise} \end{cases}$$

Figure 3. Semantics of the abstract assembly programming language (AAPL)

Definition 2 (definition position $def(p)$)**.** *Let $def(p)$ be the set of DM whose values are defined at position p in an execution trace T.*

Definition 3 (use position $use(p)$)**.** *Let $use(p)$ be the set of DM whose values are used at position p in an execution trace T.*

Definition 4 (Def-clear path)**.** *$\forall i, k \in POS$, $\underline{dm} \in DM$ and $i < k$. The path $< i, ..., k >$ is Def-clear path iff $\forall j \in < i, ..., k >$, $\underline{dm} \notin def(j)$.*

Definition 5 (recent definition position $dp(\underline{dm}^i)$)**.** *For an execution trace T, let $i \in POS_T$ and $\underline{dm}$ be a DM in T. The function $dp(\underline{dm}^i)$ computes the position of most recent data definition of $\underline{dm}$ with respect to any given point, i, in T. $dp(\underline{dm}^i) = k$ iff $\exists < k, ..., i >, \underline{dm} \in def(k)$ and $< k + 1, ..., i >$ is Def-clear path or $k = 0$ (no definition exists for $\underline{dm}^i$).*

The most recent definition of DM can be computed as a program executes by updating the *recent definition position* of DM. $dp(DM)$ allows one to keep track of positions of state nodes which define program DM in a trace. For instance, if a data manipulator $\underline{dm}$ is defined at position $i \in POS_T$ in T then for a given position $j \in POS_T$ where $i < j$, $dp(\underline{dm}^j)$ represents the position i of that state.

Definition 6 (dynamic data dependence $s_i \overset{ddd}{\rightarrow} s_j$)**.** *In an execution trace T, let $i, j \in POS$ where $i < j$ and $s_i, s_j \in T$. s_j is (directly) data dependent on s_i iff:*

1. *there exists a data manipulator $\underline{dm} \in DM$ in T such that $\underline{dm} \in use(j)$, and*
2. *$dp(\underline{dm}^j) = i$*

Note that condition 2 in Definition 6 ensures that $\underline{dm}$ is not redefined after position i in the trace. Due to Definition 5 capturing most recent definitions of data manipulators during in the execution trace.

During an execution of a program with a test input, data dependence edges and a *dynamic* data dependence graph can be constructed from an execution trace and information gathered at runtime.

Definition 7 (data dependence edge DE)**.** *A data dependence edge is an ordered pair of positions of program states in an execution trace. A directed edge DE is constructed between a pair of positions of state nodes in an execution trace, s.t. $DE = (j, i)$, iff $s_i \overset{ddd}{\rightarrow} s_j$.*

Definition 8 (dynamic data dependence graph ($DDDG$))**.** *a dynamic data dependence graph is a set of data dependence edges DE which represents the data dependencies between state nodes in a program execution trace.*

Since we are interested in slicing execution traces of assembly code, the definitions below capture the notion of the semantic trace slice.

Definition 9 (trace slicing criterion (TSC))**.** *A trace slicing criterion of an AAPL program P executed on program input (test case) tc is a pair, $TSC = (tc, dm, k)$, where dm is a set of program data manipulators after the execution of program state at position k in a program execution trace.*

Definition 10. *$DDDG_{TSC}$ is the set of data dependence edges obtained from $DDDG$ by computing backward reachability in $DDDG$ from the position specified by $dp(\underline{dm}^k)$ for each $\underline{dm}$ in the trace slicing criterion TSC.*

Definition 11 (semantic trace slice (STS).)**.** *A semantic trace slice of an AAPL program execution trace T is an execution trace T' which is a projection of T relevant to the value of the slicing criterion dm. That is, T' is T state nodes not in $DDDG_{TSC}$ left out.*

An important property of STS is that it preserves the effect of the original program execution trace on the data manipulator chosen at the selected point of interest within the trace. Although any static data slice of a program can be computed by a pure static analysis, the computation of dynamic data slice requires runtime information. The runtime information is generated as the program is executed with a given program input. This dynamic information provides the control flow path the program follows to reach the specific state of the program command in the slicing criterion. Definition 11 captures the set of all reachable program execution states from position k in the execution path T that directly or indirectly affects data manipulator $\underline{dm}$ in TSC. Thus, the semantic trace slice preserves the program's behaviour with respect to a slicing criterion data manipulator and removes any irrelevant state nodes from T, producing a more precise slice. This definition will be applied by our algorithm.

4.2 Overview of the STS Algorithm

Dynamic slicing algorithms typically first carry out all the static computation of the control dependencies and then construct the dynamic program dependence graph (DPDG) to calculate the slice. The generated slices are program statements which may be a subset of the original program [13, 3]. Our goal is to slice execution traces of a program under inspection and to generate trace slices that are a subsequence of the original execution trace which can be used to detect malware variants. Moreover, we observe that the execution trace captures the full control flow and data manipulation information of the program's execution for a given input. Therefore, a program trace abstracts away the effect of control dependencies and we know the complete path followed during the execution in which the value of the TSC data manipulator is computed. For this reason, we propose a precise trace slicing algorithm that does not perform any static evaluation of control dependency or PDG. We refer to this algorithm as the *semantic trace slicing* (STS) algorithm.

The STS algorithm employs the dynamic definition-update analysis of the data manipulators to recover dynamic data dependencies between program execution states. When a program execution begins, the algorithm performs dynamic definition-update of all data manipulators in each executed nodes via $dp()$ function (Definition 5). This allows the algorithm to compute new data dependencies in a dynamic fashion from the execution trace and then to construct the *DDDG*.

The construction of a *DDDG* with dynamic definition-update analysis of DMs allows the production of precise dynamic slices for any data manipulators at any execution position in the trace. For example, if we need the dynamic data slice for the value of a data manipulator $\underline{dm}$ at position p in the program execution trace, we begin traversing the computed *DDDG* from the definition position of $\underline{dm}$ which is recovered from the definition-update analysis (i.e $dp(\underline{dm}^p)$). Thus, the algorithm needs to traverse the execution trace only once to compute data dependencies during the computation of any trace slice.

In essence, this algorithm produces a precise trace slice that consists of only those program execution states in the execution trace T_{tc} which contribute to the computation of the value of slicing criterion.

4.3 Description of the STS Algorithm

The STS algorithm handles execution traces of an assembly level program. For a given execution trace, it analyses an execution state by state and generates the *DDDG*. Then a slice is computed with respect to the slicing criterion.

During the execution of the program with a test case tc, the execution states of program commands are stored as *state nodes* in an execution trace T_{tc}. The STS algorithm uses data dependence relations that are established between state nodes in the program execution trace with respect to an input tc. The STS algorithm presents the notion of dynamic dependence edges for identifying trace slices. The data dependence associated with each command in a program arises as a state node of the command is created in the program execution trace. New dependence edges, DE, between program state nodes are only established when their associated dynamic data dependence exists. That is, as dependence edges are established, the *DDDG* for a particular T_{tc} is created. During the execution of a program, let assume that a *dynamic* outgoing dependence edge is established from a state node s_j at position j with already existed state node s_i in the execution trace T_{tc} (i.e, $i < j$). Then the updated *DDDG* after the execution of the state node s_j is *DDDG* $\leftarrow$ *DDDG* $\cup \{DE\}$, where $DE = (j, i)$. After constructing the *DDDG* for the execution trace T_{tc}, our STS algorithm computes the *backward* reachable subgraph with respect to any given TSC, and all state nodes that appear in the reachable subgraph are contained in the semantic trace slice. That is, the execution trace slice is computed by traversing only the relevant dynamic dependence edges in the *DDDG*.

Algorithm 1 shows the pseudo code for the semantic trace slice algorithm. It constructs the *DDDG* for a program execution trace T_{tc} via computing data dependence edges between state nodes in T_{tc}. Then the algorithm computes the trace slice. In the first step of Algorithm 1, the program is executed with input tc up to execution position k. During the execution, the algorithm computes data dependence edges for each executed command. This step is a while loop (steps 8 to 12). On each iteration of the while loop, a new state node s_j is selected and DE is computed. In step 10, the procedure find_data_dep_edge(s_j) identifies data dependence edges by finding the state node at position $dp(\underline{dm}^j)$ that define data manipulator $\underline{dm}$ which is being used at position j. The procedure is presented in more detail in steps 15 to 22. If there exists a definition position node $dp(\underline{dm}^j)$ in the execution trace T_{tc} such that $dp(\underline{dm}^j) < j$ then the procedure creates a dependence edge $DE = (j, dp(\underline{dm}^j))$ and includes it into the set *DDDG*. In steps 23 to 27, the procedure update_dp(j) updates *recent definition positions* of all data manipulators that are defined in the current state s_j such that $dp(\underline{dm}) = j$ if $\underline{dm} \in def(j)$. The process of identifying data dependencies for state nodes in a trace and creating dependence edges in *DDDG* continues until the execution trace reaches position k. Finally, the procedure compute_STS() in steps 28 to 39 performs backward slice and produces the sequence of state nodes in T_{tc} that can be reachable from the slicing criterion via DE in *DDDG*.

Example 1. *We illustrate the working of STS algorithm with the aid of the following sample AAPL program of Figure 2a and its execution trace of Figure 2b. When Algorithm 1 is applied for the program trace T_{tc} of Figure 2b for $\underline{dm} = R0$ at position 14, the DDDG and STS are computed and presented in Figure 4. The semantic trace slice for $R0$ at position 14 in T_{tc} is computed in the following way:*

After the initialisation step in compute_STS() (steps 29-30 of Algorithm 1), all nodes in the trace are set as not marked and not visited, the slice set is set empty and the algorithm marks the most recent definition node of $R0$ (i.e. $dp(R0^{14}) = 10$). After the first iteration of the while loop, $STS = s_{10}$ and the following state is set as marked and not visited in T_{tc} in step 38: $\{s_5\}$ because it is reachable from s_{10} in the DDDG of Figure 4a. After the second iteration of the while loop in step 34, the

Algorithm 1 Semantic Trace Slicing Algorithm

Input: A program P and a trace slicing criterion $TSC = (tc, \underline{dm}, k)$
Output: A semantic trace slice STS

begin
T_{tc}: a sequence of state nodes
$DE := (S, S)$: a dynamic dependence edge where
S is an index of a state node in the trace T_{tc}
$DDDG$: a set of dynamic dependence edges
Execute P on input tc up to position k:
while $T_{tc} \leftarrow$ EXE(P,tc,k) does not reach execution position k **do**
  Select current executed state s_j in T_{tc}
  find_data_dep_edge(j);
  update_dp(j);
end while
compute_STS();
end

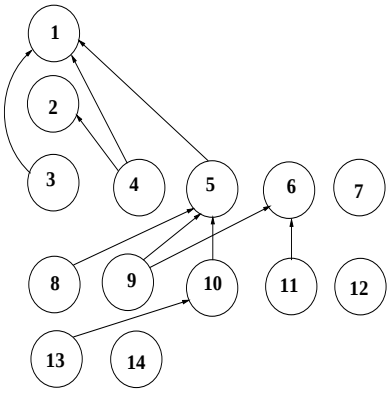
procedure find_data_dep_edge(j)
for all $\underline{dm} \in use(j)$ **do**
  if $\exists z^{dp(d^j)} \in T_{tc}$ s.t. $dp(\underline{dm}^j) < j$ **then**
    create dynamic data dependence edge $DE = (j, dp(\underline{dm}^j))$
    $DDDG \leftarrow DDDG \cup \{DE\}$
  end if
end for
end procedure

procedure update_dp(j);
for all $\underline{dm} \in def(j)$ **do**
  update definition position for $\underline{dm}$ with position j: $dp(\underline{dm}) \leftarrow j$
end for
end procedure

procedure compute_STS()
$STS \leftarrow \emptyset$
Set all state nodes in T_{tc} as not marked and not visited
Set $s_{dp(\underline{dm}^k)} \in T_{tc}$ as marked and not visited state
while there exists marked and not visited state in T_{tc} **do**
  Select marked and not visited state $s_q \in T_{tc}$
  Set s_q as visited in T_{tc} and $STS \leftarrow STS \cup \{s_q\}$
  for all outgoing dep. edges from s_q to some state s_i in $DDDG$ s.t. $DE = (q, i)$ **do**
    Find and mark $s_i \in T_{tc}$
  end for
end while
end procedure

slice contains s_{10} and s_5 is set as marked and not visited state in T_{tc} so far. The following is the outcome of the remaining while loop iterations of Algorithm 1:

- *After third iteration: $STS = < s_5, s_{10} >$ and marked and not visited nodes in $T_{tc} = \{s_1\}$.*
- *After fourth iteration: $STS = < s_1, s_5, s_{10} >$ and marked and not visited nodes in $T_{tc} = \{\phi\}$.*



(a)

STS :

s_1	R0:=n, (ρ_{s_1}, m_{s_1})
s_5	R0:=R0+1, (ρ_{s_5}, m_{s_5})
s_{10}	R0:=R0+1, $(\rho_{s_{10}}, m_{s_{10}})$

(b)

Figure 4. (a) $DDDG$ of the program in Fig.2a with respect to its T_{tc} in Fig.2b. (b) STS for $R0$ at position 14 in Fig. 2b computed by Algorithm 1.

4.4 Correctness Proof

In this section we prove that our algorithm produces a correct trace slice STS for a given program execution trace $T \in S^*$ with respect to TSC. Recall from the semantics of AAPL in Section 3, that the transition function $\hat{\mathbf{C}}$ transforms state s into s' if $\hat{\mathbf{C}}[\![c]\!]\xi = s'$ where $s = (c, \xi)$ and $s' = (c', \xi')$. Since our slicing notion handles program execution traces and produces trace slices, the definitions below introduce labeled transitions with the label identifying the *sliced* program state in T (if any):

- $T \vdash s \xrightarrow{s} s'$ if $\hat{\mathbf{C}}[\![c]\!]\xi = s'$ and $s = (c, \xi) \in STS$;
- $T \vdash s \xrightarrow{\tau} s'$ if $\hat{\mathbf{C}}[\![c]\!]\xi = s'$ and $s = (c, \xi) \notin STS$;
- $T \vdash s \overset{\tau}{\Rightarrow} s'$ for the reflexive transitive closure of $T \vdash s \xrightarrow{\tau} s'$;
- $T \vdash s \overset{s}{\Rightarrow} s'$ if $\exists s'_1$ s.t. $T \vdash s \xrightarrow{s} s'_1$ and $T \vdash s'_1 \overset{\tau}{\Rightarrow} s'$.

That is, if the execution trace can have sliced program states then these states appear in the trace slice, but not the vice versa. In the following definitions, we present the notion of a weak simulation relation between two program states. First, we define the *slice successor* label which describes the transition between the sequence of sliced states in an execution trace.

Definition 12 (slice successor $\rightsquigarrow$)**.** *Let $s = (c, \xi)$, and $s'' = (c_{s''}, \xi_{s''})$ where $s, s'' \in T$. s'' is the successor of s in the trace T, $s \rightsquigarrow s''$ iff $s, s'' \in STS$ and one of the following must hold:*

1. *$T \vdash s \xrightarrow{s} s''$ (i.e. s'' is the immediate child of s) or,*
2. *$T \vdash s \overset{s}{\Rightarrow} s''$, and $\exists s'$ s.t. $s' \notin STS$, $T \vdash s \xrightarrow{s} s'$ and $T \vdash s' \overset{\tau}{\Rightarrow} s''$.*

Note that for each sliced program state in an execution trace, there exists only one slice successor state. A definition of *weak simulation* between two program states in two different execution traces is introduced below. The definition uses a general notion of states transition $\Rightarrow$ between program states in a trace.

Definition 13 (weak simulation)**.** *A binary relation $\diamond$ is a weak simulation if $\exists s \in T_1$ and $\exists n \in T_2$ and whenever $s \diamond n$ and $T_1 : s \Rightarrow s'$ then $\exists n'$ s.t. $s' \diamond n'$ and $T_2 : n \Rightarrow n'$.*

We define the notion of relevant data manipulators in an execution trace which will be used for a weak simulation relation R and the correctness proof.

Definition 14 (relevant data manipulators)**.** *Let $i \in POS_T$ in a trace T and $s_i = (c_{s_i}, \xi_{s_i}) \in T$. We define $RDM(i)$, the set of relevant data manipulators at position i such that $\underline{dm} \in RDM(i)$ iff $\exists k \in POS_T$ and there exists a path, $< i, ..., k > \in T$ s.t. $\underline{dm} \in use(k)$, and $< i, ..., k-1 >$ is a Def-clear path wrt $\underline{dm}$.*

That is, the notion of $RDM()$ describes the set of data manipulators that are not re-defined along a particular sequence of states in a trace. In the following definition, we present a relation R and later show it is a weak simulation in Theorem 1 if the trace slice STS is closed under $\overset{ddd}{\rightarrow}$. That is, let $s, m \in T$, if $s \overset{ddd}{\rightarrow} m$ and $m \in STS$ then $s \in STS$.

Definition 15 (relation R)**.** *Let $s_i = (c_{s_i}, \xi_{s_i}) \in T$ and $n_j = (c_{n_j}, \xi_{n_j}) \in STS$. We define $s_i \; R \; n_j$ to hold iff:*

1. $c_{s_i} = c_{n_j}$
2. $\exists \underline{dm} \in RDM(i), \rho_{s_i}(\underline{dm}) = \rho_{n_j}(\underline{dm})$ *and* $m_{s_i}(\underline{dm}) = m_{n_j}(\underline{dm})$.

That is, the relation R holds between a state s_i of the original execution trace and a state n_j of the trace slice whenever s_i is a sliced state and it corresponds to the sliced state n_j in the trace slice.

Example 2. *Consider program states $s_{10} \in T_{tc}$ in Fig.2 and $s'_3 \in T'_{tc}$ in Fig.5, where $STS = \; < s_1, s_5, s_{10} >$. We have $s'_3 \in STS$, $c_{s_{10}} = c_{s'_3}$, $R0 \in RDM(10)$, $\rho_{s_{10}}(R0) = \rho_{s'_3}(R0)$ and $m_{s_{10}}(R0) = m_{s'_3}(R0)$ and hence $s_{10} R \; s'_3$.*

The following theorem presents the notion of semantic trace slice correctness. It guarantees that there is a weak simulation relation R between the original execution trace and the trace slice in particular the slice criterion TSC in both traces satisfies the relation R. We provide a proof sketch to show the correctness property.

Theorem 1. *Let $i, j \in POS_{T_1}$, $s = State(i), s'' = State(j) \in T_1$ and $x, y \in POS_{T_2}$,$n = State(x), n'' = State(y) \in T_2$. Assume that STS is closed under $\overset{ddd}{\rightarrow}$ Whenever $s \; R \; n$ and $T_1 \vdash s \rightsquigarrow s''$ then*
$\exists n''$ s.t. $s'' R \; n''$ and $T_2 \vdash n \rightsquigarrow n''$.

Proof Sketch: From $s \; R \; n$ we infer $n \in STS$ and $c_s = c_n$. We also infer that $\exists \underline{dm} \in RDM(i)$: $\rho_s(\underline{dm}) = \rho_n(\underline{dm})$ and $m_s(\underline{dm}) = m_n(\underline{dm})$. From $T_1 \vdash s \rightsquigarrow s''$ we infer $s, s'' \in STS$, so with $s'' = (c_{s''}, \xi_{s''}) \; \exists \; n'' = (c_{n''}, \xi_{n''}) \; s.t. \; T_2 \vdash n \rightsquigarrow n''$, and thus $n'' \in STS$. We now show that $s'' R \; n''$:

1. There are two cases to show that $c_{s''} = c_{n''}$:
   - $\forall \underline{dm} \in def(j), \exists E \; s.t. \; \hat{\mathbf{C}}[\![\underline{dm} := E]\!]\xi_{s''} = \hat{\mathbf{C}}[\![\underline{dm} := E]\!]\xi_{n''}$ and thus $c_{s''} = c_{n''}$.
   - if $def(j) = def(y) = \phi$, then $\exists \underline{dm} \in RDM(j)$, $\underline{dm} = use(j) = use(y)$ and $\rho_{s''}(\underline{dm}) = \rho_{n''}(\underline{dm})$ and $m_{s''}(\underline{dm}) = m_{n''}(\underline{dm})$.
2. For a given $\underline{dm} \in RDM(j)$, $\rho_{s''}(\underline{dm}) = \rho_{n''}(\underline{dm})$ and $m_{s''}(\underline{dm}) = m_{n''}(\underline{dm})$ hold in two cases:
   - if $\underline{dm} \in def(i)$, and since $c_s = c_n$ from $s \; R \; n$ then $\exists E$ s.t. $c_s = c_n = (\underline{dm} := E)$, thus $s' = \hat{\mathbf{C}}[\![\underline{dm} := E]\!]\xi_s$ and $n' = \hat{\mathbf{C}}[\![\underline{dm} := E]\!]\xi_n$ and from $s \rightsquigarrow s''$ and $n \rightsquigarrow n''$, we infer that $\rho_{s'}(\underline{dm}) = \rho_{s''}(\underline{dm})$ and $m_{s'}(\underline{dm}) = m_{s''}(\underline{dm})$ and $\rho_{n'}(\underline{dm}) = \rho_{n''}(\underline{dm})$ and $m_{n'}(\underline{dm}) = m_{n''}(\underline{dm})$, and thus $\rho_{s''}(\underline{dm}) = \rho_{n''}(\underline{dm})$ and $m_{s''}(\underline{dm}) = m_{n''}(\underline{dm})$.
   - if $\underline{dm} \notin def(i)$, then $\underline{dm} \in RDM(i)$, and the claim follows from $\forall \underline{dm} \in RDM(i) : \rho_s(\underline{dm}) = \rho_n(\underline{dm})$ and $m_s(\underline{dm}) = m_n(\underline{dm})$ since $\xi_{s''} = \xi_s = \xi_n = \xi_{n''}$ wrt $\underline{dm}$.

□

Example 3. *The trace slice computed in Fig. 4b for register $R0$ in the program trace in Figure 2 is not "executable" in the sense that it does not correspond to an execution but we can produce an "executable" program P' from the trace slice via extracting the command sequence $P' = \Sigma(C)$ from the trace slice in Fig.4b. Then, in Fig.5, we execute the program P' with the same program input ($n = 1, m = 2$) and produce the execution trace (a projection) which agrees with the original trace (in Fig. 2) on the values of the slicing criterion. So we are saying that we have the correct sub trace if we can execute the program projection from the input and agree with the original trace at the corresponding program point. Therefore, the observable behaviour of program P' trace is similar to the observable behaviour in the original program trace of figure 2 with respect to the slicing criterion.*

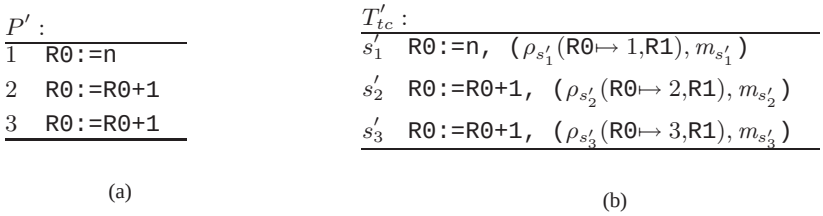


Figure 5. (a) The program P' is produced by extracting the command sequence from the trace slice in Fig. 4b; (b) an execution trace of P' on input $tc : n=1, m=2$.

5 Strengths and Limitations of the STS Algorithm

The main motivation for our STS Algorithm is to minimize the false-negative rate in detecting obfuscated malware variants. This can be accomplished by removing the effects of the obfuscation techniques (deobfuscation) and capturing the true semantics of program traces using slicing. Thus, the power of our STS algorithm relies on the ability to handle malware obfuscating transformations. We discuss below the set of obfuscating transformations that are used to generate new malware variants and STS algorithm can handle, we call this set *STS-handled* obfuscations. *STS-handled* obfuscations are code transformations that add new (syntax) code lines to create new program variants while preserving the data dependence structure of the original program. *Code reordering*: This obfuscation technique, commonly is applied on independent commands where their order in the code do not affect other commands. The execution order of commands can be maintained using unconditional jumps. Thus, new variants of the program can be created with the same semantics but different syntax. *Garbage insertion*: This transformation technique introduces commands that have no semantics effect on the program execution. The main objective of the technique is to create new program variants that preserve the original program semantics but contain different syntax. *Equivalent functionality*: This obfuscation technique replaces commands with other equivalent commands that perform the same operations of the original code. *Opaque predicate*: A predicate whose value is known a prior to a code transformation but is hard to determine by examining the obfuscated code [7]. This technique obfuscates the program control flow and makes it difficult to analyze statically.

Limitations. The STS algorithm has some limitations. The slicing algorithm is not resilient with respect to data obfuscation and variable renaming techniques. Introducing new data dependences between program registers and memory locations is an obfuscation technique which can not be handled via our slicing algorithm (*STS-unhandled obfuscation class*). This transformation technique obfuscates a program via creating dependencies between variables using rewriting assignments or introducing new ones [7, 16]. For instance, malware writers may use this technique to split a register into two registers or to transform a register $R0$ into the expression $R1 * R0 + R2$ where $R1$ and $R2$ contain dummy constant values. Thus, this technique increases the number of data dependencies in the obfuscated variant which causes to have different semantics of trace slices comparing with the malware parent trace slices. An example in Fig.6 illustrates this transformation technique. *Variable renaming* is an obfuscation technique which used by malware writers to obfuscate their code and to produce new malware variants by simply changing registers and variable names in the program. An ongoing research investigation for a possible solution to this obfuscation class is presented briefly in the following section.

6 Review of related work

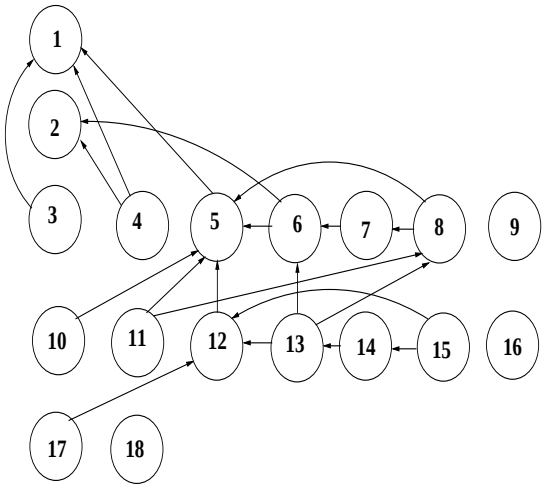
Dynamic slicing has been extended from the traditional slicing techniques for debugging programs [4] to a wider set of applications such as dynamic slicing for concurrent programs [17, 20], and software testing [15]. Dynamic slicing approach takes into the consideration only one execution history of a program to compute a slice. Thus, it may significantly reduce the size of the slice as opposed to the approach of static slicing. To present all of dynamic program slicing approaches would be out of scope of this report. A survey of dynamic program slicing techniques and applications can be found in [22, 21].

In dynamic slicing techniques that depend on an execution trace, the computed dynamic slice is a subset of the original program. Korel et al. [13, 14] extended Weiser's static slicing algorithm to the dynamic approach. They incorporated the execution history of a program as a trajectory to find the statements that actually affect a variable at a program point. Thus, the resulting slices are more compact and precise than the program slices proposed by Weiser. Agrawal and Horgan [3]

$\mathcal{O}(P)$:

```
1          R0:=n
2          R1:=m
3  Loop:   (R0 >= 3) JMP Exit
4          *R1:=F(*R0)
5          R0:=R0+1
6          R1:=R0+R1
7          R1:=R1+1
8          R1:=R1-R0
9          JMP Loop
10 Exit:   JMP ...
```

(a)



(b)

Figure 6. An obfuscated code variant of program P in Fig.2 and its $DDDG$ after applying data obfuscations.

provided a novel approach for computing dynamic program slices via program dependence graphs (PDGs). Their algorithm uses the reduced dynamic dependence program (RDDP) where a new node is created if it introduces a new dependence edge with other existing nodes in RDDP. However, different occurrences of the same node cannot be distinguished in RDDP. None of the above mentioned slicing methods provide a way to capture the runtime values of variables in a program slice without at least re-executing the slice. On the contrary, our approach extracts a dynamic trace slice from an execution history. The computed slice preserves the semantics of the original program execution trace.

Zhang et al. [23, 24] present a dynamic slicing technique that depends on a recorded execution history. Their limited preprocessing (LP) algorithm performs some preprocessing to first augment the record with summary information and then it uses demand driven analysis to extract dynamic dependences from the augmented record. In this sense, our approach is similar to the approach of Zhang et al. In our approach, the data dependence information is computed on-the-fly during the program execution and is not used to augment the execution trace, but it is mainly used to construct the DDDG.

In terms of slicing binary executables, it is hard to find practical slicing solutions for binary executable programs in the literature. The existing techniques proposed in the literature perform static slicing only. Cifuentes and Fraboulet use intraprocedural slicing for handling indirect jumps and function calls in their binary translation framework [6]. Debray et al. [11] and Kiss et al. [12] presented methods for the interprocedural static slicing of binary executables. However, these approaches requires extracting static data dependence information from a CFG. On the contrary, our algorithm does not rely on a CFG but it computes these information from a program execution trace. Bergeron et al. [5] propose a static slicing technique for analyzing assembly code to detect malicious behavior. Their approach compares program slices against behavioral specifications (e.g. a set of API signatures) to detect potentially malicious code. However, since their method is purely based on signatures of function calls and sequence of commands, it lacks the ability to handle certain obfuscation techniques such as code reordering and equivalent functionality.

7 Mapping Semantic Traces

The objective of the mapping process is to automatically identify a correspondence between *executed program states* (nodes) from the two semantic traces. The two semantic traces are produced by collecting the execution traces of two program variants. We assume that one variant of the program created after applying semantics preserving program transformations [7, 8]. In establishing a map between a pair of semantic execution traces it is our objective to provide an algorithm which produces *complete* and *correct* results. That is our algorithm finds as many true mappings as possible (*completeness*) and it

finds only true mappings.
Our method needs a pair of execution traces for two variants of a program, it establishes a correspondence between the executed states by examining the *semantic* details of individual states in both execution traces. The mapping process consists of three main steps:

- **State matching**. For each given program state in the trace, the semantic value is produced i.e. the *execution context*. The semantic values are used to compare two program states and to identify potential mappings or exclude mappings and states throughout the next two steps.
- **Redundant abstraction**. Our matching method begins by examining each execution trace of two program variants for any *redundant* program states. This process abstracts away program states that contain similar semantic details of already executed states in the trace. The outcome of this step is an ordered sequence of *semantically* unique program states of a given execution trace. Algorithm 2 outlines the redundant abstraction procedure.
- **Trace mapping**. Given a pair of ordered sequences of unique execution states, an iterative algorithm is used to establish mappings between the states (nodes). For each state in the first sequence, the algorithm identifies a correspondence candidate state in the other sequence.

Next we discuss the details of how the *semantic* values of execution states are used in the state matching step. Then we discuss the details of redundant abstraction and trace mapping algorithms.

Label	Category	Obfuscation
gi	Garbage insertion	$\{\} \rightarrow \{C\}$
eo	Equivalent operation	$\{op\} \rightarrow \{\overline{op}\}$
op	Opaque predicate	$\{\} \rightarrow \{P^{T/F}\}$
rr	Register renaming	$\{Rx\} \rightarrow \{Ry\}$
cs	Command split	$\{C\} \rightarrow \{C_x, C_y\}$
cm	Command merging	$\{C_x, C_y\} \rightarrow \{C_{xy}\}$
cr	Command reorder	$\{(C_x, C_y)\} \rightarrow \{(C_y, C_x)\}$

Table 1. Obfuscating transformations.

7.1 State Matching

Before we present our algorithm of mapping a pair of execution traces of two program variants, we introduce the matching step between a pair of execution states. The mapping algorithm establishes mappings between two execution traces based on the successful matches of execution states. When the *state matching* step matches a pair of execution states, it essentially compares the *semantic* values produced by both states. The *semantic* values produced by an execution state can either represent a set of *environment* values or a set of *memory* values. Since our mapping step deals with execution traces of obfuscated program variants, program syntax, i.e. *commands*, may be altered and also some program variables may be replaced with different ones. Thus, establishing an exact match between execution states is unlikely to succeed. Therefore our *state matching* step uses the results computed from individual instructions and ignores commands syntax such that the derived *semantic* results can be easily matched even if program obfuscations have affected the corresponding instructions. For execution traces with long execution state sequences, it is unlikely to map traces based on semantic results of execution states that do not correspond to each other. However, there is a chance of false (i.e. *coincidental*) mappings between a pair of execution traces with very short execution state sequences. Thus, to avoid such false mappings, our state matching method consists of the following *semantic* components:

- **Environment values (EV)**. To match environments of execution states, the environment values are extracted from execution states and represented in single values. Each environment of an execution state returns a single value (EV) which represents the *evaluated data* value of a data manipulator at that particular execution state. When matching a pair of execution states, we look for a match in the evaluated data values of both states. Given two execution states s_1 and s_2 with their data values EV_1 and EV_2, respectively. We consider s_1 matches s_2 if the value of EV_1 is similar to the value of EV_2.

Algorithm 2 Redundant Abstraction

```
1  Input: a semantic trace ST
2  Output: A non-redundant nodes list worklist
3  redundantlist: a redundant list of ST nodes
4  redundancy_abstraction(ST){
5  worklist ← ST
6  i ← first_index(worklist)
7  while worklist ≠ φ do
8     j ← i + 1
9     while n_j ≠ ⊥ do
10       if state_matching(n_i,n_j) then
11          redundantlist ← redundantlist ∪ {n_j}
12          worklist ← worklist − {n_j}
13       end if
14       j ← next_j^th_index(worklist)
15    end while
16    i ← next_i^th_index(worklist)
17 end while
18 return(worklist)
```

- **Memory values**. When we match memories of execution states, it is unlikely to find *true* matches of memory addresses between execution states of both variants. That is because memory locations of two program variants may vary at runtime. Also, matching the offsets of memory address of both variants may not be effective in finding matches because assume that programs might incorporate dynamic code generation and code reordering techniques to execute new code with different memory layout (i.e. offset). Therefore, memory values are used to establish matches between corresponded execution states instead. Memory addresses are only used to obtain the memory values MV of data manipulators in execution states where memory updates have been performed. The memory match step is performed between a pair of execution states that have updated memory values. The comparison of MVs can be performed in the same fashion as that for EVs.

7.2 Trace Mapping

This section describes the trace mapping algorithm and how the algorithm establishes mappings between a pair of execution traces of two program variants. As stated in the introduction section, that the goal is to map two trace variants of a program where another program variant may has been produced via some program transformations (*obfuscation*). Semantic preserving program obfuscations can have significant affects on program syntax, i.e. program commands. In particular, obfuscating transformations may *rename* program registers, *add* irrelevant commands to the original program, e.g. garbage and opaque predicate commands, or some transformations may *split*, *reorder* or *merge* commands. Table 1 contains some code transformation techniques deployed in creating new program variants.

An example in Figure 7, illustrates the above obfuscation affects on program syntax. In this figure each program command is labeled by a letter. New commands that have been introduced in the program variant are labeled by the obfuscations labels that have been used to create these commands. Corresponding commands in the original and obfuscated variants are labeled with a and a', respectively. We use subscripts to show the correspondence between one command in one variant and multiple instructions in the other variant.

Fig.3 presents our trace mapping algorithm which has been developed to identify mappings between a pair of execution traces of two variants of a program under the presence of the above mentioned obfuscating transformations.

8 Test Data Generation via Dynamic Domain Reduction (DDR)

The objective of the test data generation analysis is to compute the set of constraints in the initial domains of program inputs which is consistent with state update and leads to final input domains. The Dynamic Domain Reduction (DDR)

P :	
a	R0:=n
b	R1:=m
c	R2:=R1
d	R3:=R2+R0
e	R4:=R1+k
f	R5:=1

(a)

P' :	
a'	R0:=n
cr_1	JMP rr_1
gi_1	R22:=R22+1
op_1	P^T JMP cm
rr_1	R11:=m
gi_2	R22:=R22+1
cr_2	JMP op_1
cm	R3:=R11+R0
e'_1	R4:=k
e'_2	R4:=R4+R11
rr_2	R15:=1

(b)

Figure 7. A sample program (a) and its variant (b) after applying program obfuscation techniques in table 1.

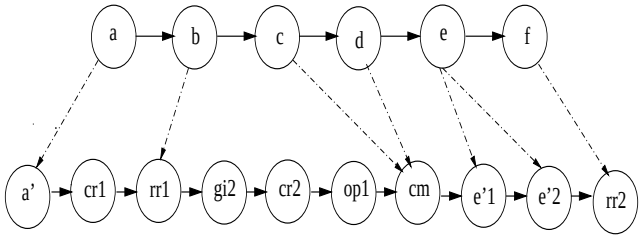


Figure 8. Mapping execution states of execution trace variants.

analysis consists of two steps:

- Forward dynamic domain reduction analysis
- Backward domain substitution analysis

The objective of the Dynamic Domain Reduction process [18] is to automatically identify and generate the reduced value domains of program inputs which represent the program outputs.

Example 4. *Let's consider the path: 123 of the code below:*

1. z=x+100
2. if ($z < 20$*)* {
3. $y = 1$;
else
4. $y = 2$;
}

Let assume the initial value domains of program inputs are $x := y := z := [-127, +128]$. *The dynamic domain reduction*

Algorithm 3 Mapping Semantic Traces

```
1  Input: a pair of execution traces ST_a and ST_b
2  Output: a list of pairs of mapped execution states mappedlist
3  begin
4  worklistA: an ordered list of unique execution states
5  worklistB: an ordered list of unique execution states
6  perform Redundancy Abstraction process on both traces ST_a and ST_b
7  worklistA ← redundancy_abstraction(ST_a);
8  worklistB ← redundancy_abstraction(ST_b);
9  set all elements in worklistA as unvisited
10 i ← first_index(worklistA)
11 j ← first_index(worklistB)
12 while n_i ≠ ⊥ do
13    if n_j ≠ ⊥ then
14       if state_matching(n_i, n_j) then
15          mappedlist ← mappedlist ∪ {(n_i, n_j)};
16          worklistA ← worklistA − {n_i};
17          worklistB ← worklistB − {n_j};
18          i ← next_index(worklistA)
19          j ← first_index(worklistB)
20       else
21          j ← next_index(worklistB)
22       end if
23    else
24       set n_i from unvisited to unmapped node in worklistA
25       i ← next_index(worklistA)
26       j ← first_index(worklistB)
27    end if
28 end while
29 end
```

analysis would produce the set of domain constraints as follow:

$$
\begin{array}{lll}
-127 < x < 128 \wedge -127 < y < 128 \wedge -127 < z < 128 & & \textit{(FC1)} \\
-127 < x < 128 \wedge -127 < y < 128 \wedge \ -27 < z < 228 & & \textit{(FC2)} \\
-127 < x < 128 \wedge -127 < y < 128 \wedge \ -27 < z \ < 20 & & \textit{(FC3)} \\
-127 < x < 128 \wedge \qquad\quad y \ = 1 \wedge \ -27 < z \ < 20 & & \textit{(FC4)}
\end{array}
$$

In Example 4, after each program command evaluation, the DDR step updates the domains of program inputs. In particular, the analysis has performed three domain updates till the final domain of program inputs is produced. The final output of DDR analysis is represented in the form of a set of constraints (line 4).

8.1 Backward domain substitution analysis (BDS)

We propose our *Backward domain substitution* approach which uses the set of reduced domains generated via the forward analysis and computes the subset of values of program inputs. The BDS analysis starts backward from the set of final constraints of program inputs and computes the next constraint. The inverse function of each program assignment command is computed and used to evaluate the set of constraints of program inputs. The BDS analysis is a bottom-up approach where the last command of the program is evaluated first using the output constraint of DDR analysis. Then the analysis propagates with new computed *backward* constraints of input domains till the start of the program is reached. The output of BDS analysis is a subset of input domains whose values can exercise the analysed program path.

Two types of program commands are considered during the BDS analysis: *assignment update* and *condition check*. The first type of commands defines and updates values of program variables. While the second type of commands evaluates a predicate and based on the evaluation outcome the control flow is determined. Thus, a new *backward* constraint can be computed as follows:

- Assignment update: the evaluation of the inverse function of the assignment update with the last backward constraint forms the new backward constraint.
- Condition check: the new backward constraint is computed by taking the intersection of the last backward constraint before the predicate statement and the forward constraint after the predicate. .

Example 5. *Let's compute a possible input domain values which exercises the path: s123456e for the following program using both the forward and backward techniques:*

s. input(x,y)
1. if ($x <= 90$*)* {
2. *if(*$y <= 15$*)* {
3. $x++;$
}
}
4. if($x == 91$*)* {
5. $y = 20;$
6. $x = 100;$
}
e. output(x,y);

Forward Analysis (Domain Reduction):

`s.`

$$\{(x, [-127, 128]), (y, [-127, 128])\}$$

`1.`

$$\{(x, [-127, 90]), (y, [-127, 128])\}$$

`2.`

$$\{(x, [-127, 90]), (y, [-127, 15])\}$$

`3.`

$$\{(x, [-126, 90]), (y, [-127, 15])\}$$

`4.`

$$\{(x, [91, 91]), (y, [-127, 15])\}$$

`5.`

$$\{(x, [91, 91]), (y, [20, 20])\}$$

`6.`

$$\{(x, [100, 100]), (y, [20, 20])\}$$

Backward Analysis (Domain substitution):

Recall that the BDS analysis starts from the last statement in the path (i.e. line e) and finishes in the first statement (i.e. line s), the input of the Backward *analysis is the computed domain in the forward analysis:*

`e.` $\{(x, [100, 100]), (y, [20, 20])\}$

`6.` $\{(x, [-127, 128]), (y, [20, 20])\}$

`5.` $\{(x, [-127, 128]), (y, [-127, 128])\}$

`4.` $(FC3 \cap BC5)\ \{(x, [91, 91]), (y, [-127, 15])\}$

`3.` $\{(x, [90, 90]), (y, [-127, 15])\}$

`2.` $\{(x, [90, 90]), (y, [-127, 15])\}$

`1.` $\{(x, [90, 90]), (y, [-127, 15])\}$

8.2 Abstract Interpretation

Abstract Interpretation [9, 10] defines the approximation correspondence between the concrete semantics $\mathsf{C}[\![\mathtt{P}]\!]$ of a syntactically correct program $\mathtt{P} \in \mathbb{P}$, where $\mathbb{P}$ is a given programming language, and an abstract semantics $\mathsf{A}[\![\mathtt{P}]\!]$ which is a safe/sound approximation of the concrete semantics $\mathsf{C}[\![\mathtt{P}]\!]$.

The abstract analysis of a program P is a *symbolic* interpretation of this program, using abstract values instead of concrete values (i.e. semantics). An abstract value represents a set of concrete values or properties of such a set. Let Z_c be the *concrete* semantic domain which is a poset $\mathcal{PS}(\mathrm{Z}_c, \sqsubseteq_c)$, i.e. partially ordered by the approximation ordering $\sqsubseteq_c$. The abstract semantics Z_a is also a poset $\mathcal{PS}(\mathrm{Z}_a, \sqsubseteq_a)$ which is partially ordered by the abstract ordering $\sqsubseteq_c$.

8.2.1 Backward Abstract Domain Interpretor

The backward abstract domain interpretor starts in a reverse way, by proceeding from the last command in a path, with the output *store list* $\overline{\Sigma}_{fout}$ of the forward abstract domain interpretor on all possible paths. For each of the different types of commands, we have described a transformation which specifies the new store(s) $\overline{\Sigma}'$ for the next command c' in the selected path. The algorithm essentially performs applications of these transformations until all stores are stabilised and the data values of program variables are generated.

Semantics rules:

$f_c^{-1} : \text{Cmd} \rightarrow \Im(\text{Cmd})$

$f_c^{-1}[\![x := e]\!]\Sigma := \Sigma' \quad$ where,

$\Sigma' = \{\sigma[var \mapsto \{v\}] \mid \sigma \in \Sigma \quad \wedge \sigma[x \mapsto val_x] \wedge var \in Var(e) \cup x \wedge v \in evaluate\_assignment(e, val_x, x)\} \cup \Sigma$

$f_c^{-1}[\![c_1; c_2]\!]\Sigma = f_c^{-1}[\![c_1]\!] \circ f_c^{-1}[\![c_2]\!]\Sigma$

$f_c^{-1}[\![\text{if } b\ c_1\ c_2]\!]\Sigma = \Sigma' \quad \text{where } \Sigma' = \begin{cases} f_b^{-1} \circ f_{c_1}^{-1}\Sigma & \text{if } b = \text{true} \\ f_{\neg b}^{-1} \circ f_{c_2}^{-1}\Sigma & \text{if } b = \text{false} \end{cases}$

$\hat{f}^*_{\text{while } b \text{ do } c} = f_c^{-1}[\![\text{while } b\ c]\!]\Sigma = \mathcal{W} \circ f_b^{-1}[\![\neg b]\!]\Sigma \quad \text{where } \mathcal{W} = lfp\lambda\chi. \cup f_b^{-1}[\![b]\!](f_c^{-1})[\![c]\!]\chi)$

Figure 9. Backward Rules for the Semantics in Fig. ??

8.3 Backward Rules

This section presents rules applied within the backward step for computing domains of values for program variables. First the semantics *backward* rule for Boolean expressions is introduced then the semantics *backward* rules for commands are presented. Later in this section the set of *backward* rules for the arithmetic expressions are presented in the sketch Algrithm **??**The function $evaluateInverseExp$ in Algorithm will be presente.

8.3.1 *Backward* Rule for Boolean Expressions.

When computing the backward values of domains, the *backward* rule for a Boolean expression propagates the values of variables for which the Boolean expression b holds to the next store list Σ'. Let denote the store list after evaluating the Boolean expression b in the *Forward Analysis* with $\sigma_b \in \Sigma_{FA}$ where σ_b contains the domains of variables for which the Boolean b holds. Also, let denote the current store list in the *Backward Analysis* with $\delta \in \Sigma$. In order to compute the right domains of variables for the Boolean expression b in the backward analysis, the domains of variables in the current store, Σ, are updated via intersecting the domains in σ_b with the store δ. Thus, the *backward* rule for Boolean expressions ensures that the variables associated with the Boolean expression b are updated with values that satisfy the Boolean b. This rule is formally defined as:

$$f_b^{-1}[\![b]\!]\Sigma = \Sigma' \quad \text{where } \Sigma' = \{\sigma_b \in \Sigma_{FA} \cap \delta_c \in \Sigma\} \cup \Sigma$$

8.3.2 *Backward* Rules for Commands

- **Assignment:** Algorithm **??** presents the set of rules which handles the *backward* computation of an assignment command $x := e$ where the domain value of an expression e needs to be updated/modified with respect to the domain value of the variable x. The procedure computes the backward domain values for x and for any variables var referenced in expression e.

$$\begin{array}{l} f_c^{-1}[\![x := e]\!]\Sigma := \Sigma' \qquad \text{where,} \\ \Sigma' = \{\sigma[var \mapsto \{v\}] \mid \sigma \in \Sigma \wedge\ \sigma[x \mapsto val_x] \wedge var \in Var(e) \cup x \wedge \\ \qquad\qquad\qquad\qquad v \in evaluate\_assignment(e, val_x, x)\} \cup \Sigma \end{array}$$

- **Sequential:** The backward rule of sequential commands, e.g., $c_1; c_2$ processes the later command(s) first (as opposed to the order taken in Forward Analysis). The rule is defined as:

$$f_c^{-1}[\![c_1; c_2]\!]\Sigma = f_c^{-1}[\![c_1]\!] \circ f_c^{-1}[\![c_2]\!]\Sigma$$

- **Conditional:** The backward rule for the conditional commands processes one of the branches (true or false branch) in the reverse order. The command(s) of the conditional command is already determined during the Forward Analysis, thus, the rule treats the branch as a sequential command which composed of a Boolean expression and a command i.e. $\hat{f}_c^*[\![b; c]\!]$:

$$f_c^{-1}[\![\text{if } b\ c_1\ c_2]\!]\Sigma = \Sigma' \quad \text{where } \Sigma' = \begin{cases} \hat{f}_c^*[\![b; c_1]\!] = f_b^{-1} \circ f_{c_1}^{-1}\Sigma & \text{if } b = \text{true} \\ \hat{f}_c^*[\![\neg b; c_2]\!] = f_{\neg b}^{-1} \circ f_{c_2}^{-1}\Sigma & \text{if } b = \text{false} \end{cases}$$

- **While Loop:** The backward rule for a while loop command evaluates the *backward* domains of program variables that are referenced in the loop Boolean expression b and all variables that are re-defined and referenced inside the body of the loop i.e. the sequence of commands within the loop c. The rule simply starts the evaluation backwards from the Boolean expression $\neg b$ which exists the body of the loop and processes the sequence of commands $b_i; c_i$ for each loop iteration i generated via the Forward Analysis.

  Example 6. *Consider the following program which only consists of a while loop command:*

  while b do c;

where b is a Boolean expression and c is a program command. Let assume that the body of the loop get executed k times during the Forward Analysis, thus the generated sequence of commands would look like the following:

$b_1; c_1$	1^{st} *iteration of the while loop*
$b_2; c_2$	
$\vdots$	
$b_k; c_k$	k^{th} *iteration of the while loop* ($i = k$)
$\neg b_{k+1}; skip$	$k+1$ *iteration exists the loop* ($i = k+1$)

In the Backward Analysis, the rule of the while loop command starts from the last command executed i.e. $\neg b_{k+1}$ and finishes after evaluating the commands $b_1; c_1$. Thus, for this particular example, the backward rule is defined as:

$$\hat{f}_c^*[\![\textit{while } b \textit{ do } c]\!]\Sigma = \hat{f}_b^*[\![b_i]\!](\hat{f}_c^*[\![c_i]\!]) \circ \hat{f}_b^*[\![\neg b_{k+1}]\!]\Sigma, \textit{ where } i = k, k-1, k-2, \ldots, 1$$

Algorithm 4 Backward Assignment Evaluation Rules

procedure: `evaluate_assignment(`e, val_e, x`)`
input : $''x := e''$: An expression e to be evluated, its value i.e., $f^*[\![e]\!] = val_e$, and the variable x
output : Evaluates the set of input values Val of x

switch e **do**
  case n
    $x \mapsto \{Id_o\}$; //where Id_o is the set of initial values in the initial store list Σ_o. **return**;
  end
  case var
    if $var == x$ **then**
      $x \mapsto \{Id\}$; // where Id is the values of x in the current store, i.e. no updates required for x.
      return;
    end
    else if $var \neq x$ **then**
      $x \mapsto \{Id\}$;
      $var \mapsto \{val_x |\ \sigma[x \mapsto val_x] \wedge \sigma \in \Sigma\}$;
      //where val_x is the set of values of x in the current store.;
      return;
    end
  end
  case e_1 *op* e_2
    switch e_1 *op* e_2 **do**
      case n *op* n
        $x \mapsto \{Id_o\}$;
        return;
      end
      case $(n$ *op* $var) \,||\, (var$ *op* $n)$
        $val_{var} \mapsto evaluateInverseExp(val_x, val_n, \mathsf{op})$;
        if $var \neq x$ **then**
          $val_x \mapsto \{Id_o\}$;
        end
      end
      case $Lvar$ *op* $Rvar$
        if $Lvar == Rvar$ **then**
          compute the domain of var: $x := var$ op var;
          $val_{Lvar} \mapsto compute\_var\_dom(val_x, Lvar, \mathsf{op})$;
          if $x \neq Lvar$ **then**
            set domain of x: $val_x \mapsto \{Id_o\}$;
          end
        end
        else
          propagate val_x to var_l and var_r for: $x := var_l$ op var_r:
          if *value domains of both* $Lvar$ *and* $Rvar$ *in FA step before current command are set to some constant numbers* **then**
            no need to propagate val_x into the variables, $Lvar$ op $Rvar$. $val_x \mapsto \{Id_o\}$;
          end
          else if val_{Lvar} *in FA step before current command is a constant number* $[n, n] \wedge val_{Rvar}$ *is not a constant number* **then**
            propagate val_x into $Rvar$ only: $val_{Rvar} \mapsto evaluateInverseExp(val_x, val_{Lvar}, \mathsf{op})$;
            if $x \neq Rvar$ **then**
              $val_x \mapsto \{Id_o\}$;
            end
          end
          else if val_{Rvar} *in FA step before current command is a constant number* $[n, n] \wedge val_{Lvar}$ *is not a constant number* **then**
            propagate val_x into $Lvar$ only; $val_{Lvar} \mapsto evaluateInverseExp(val_x, val_{Rvar}, \mathsf{op})$;
            if $x \neq Lvar$ **then**
              $val_x \mapsto \{Id_o\}$;
            end
          end
          else if *value domains of both* $Rvar \wedge Lvar$ *in FA step before current command are NOT set to some constant numbers* **then**
            split_domain($Lvar$ op $Rvar$,val_x,val_{Lvar},val_{Rvar});
            if $x \neq Lvar \wedge x \neq Rvar$ **then**
              $val_x \mapsto \{Id_o\}$;
            end
          end
        end
      end
      See Algorithm 5 for details of the cases below:
      case $(e$ *op* $var) \,||\, (var$ *op* $e)$
      end
      case $(e$ *op* $n) \,||\, (n$ *op* $e)$
      end
      case e_l *op* e_r
      end
    end
  end
end

Algorithm 5 Continuation of evaluate_assignment procedure in Algorithm 4

continue-procedure: `evaluate_assignment(`e, val_e, x`)`

case $(e$ *op* $var)$ || $(var$ *op* $e)$
  if val_{var} *is constant in FA step before current command* **then**
    $val_e \mapsto evaluateInverseExp(val_x, val_{var}, \mathsf{op})$;
    `evaluate_assignment(`e, val_e, x`)`;
  end
  else
    if $\forall v \in ref(e), val_v$ *is constant in FA before current command* **then**
      $val_e \mapsto [\![e]\!]\Sigma_{\text{FA}}$;
      $val_{var} \mapsto evaluateInverseExp(val_x, val_e, \mathsf{op})$;
    end
    else
      split_domain(e op var,val_x,val_e,val_{var});
      `evaluate_assignment(`e, val_e, x`)`;
    end
  end
end
case $(e$ *op* $n)$ || $(n$ *op* $e)$
  $val_e \mapsto evaluateInverseExp(val_x, val_n, \mathsf{op})$;
  `evaluate_assignment(`e, val_e, x`)`;
end
case e_l *op* e_r
  split_domain(e_l op e_l,val_x,val_{e_l},val_{e_r});
  `evaluate_assignment(`e_l, val_{e_l}, x`)`;
  `evaluate_assignment(`e_r, val_{e_r}, x`)`;
end

9 Conclusion

In this report, we have introduced our new algorithm to slice execution traces with a sketch of the correctness proof. The algorithm supports the process of capturing semantic details of trace slices for detecting obfuscated malicious code. The slicing in this context has two roles: to reverse engineering the effect of obfuscations and to produce smaller semantic traces of suspicious program executions for matching. We have introduced a simple programming language (AAPL) which provides a high level imperative representation of the assembly code. Also, the semantic trace mapping approach and the backward domain reducation technique in finding test data inputs have been presented and their components discussed briefly. The describtion of the semantic trace mapping algorithm and the test data generation technique using backward domain reduction have been presented. The trace matching algorithm uses sliced traces to finding possible mappings between execuation states. The proposed test data generation technique helps to approximate the set of possible input values at each program point. These can be used to find exact test inputs with aid of search-based techniques.

Our preliminary experience has shown that the trace slicing algorithm can be of a great help for malware detectors during the process of matching obfuscated malicious program variants. However, more research and experimentation is needed to better understand the advantages and limitations of our slicing and mapping algorithms in handling more advanced code transformation techniques. So far we have performed experiments with programs that have been transformed via the *STS-handled* obfuscations (see Section 5). We are planning to perform experiments on obfuscated programs with *STS-unhandled* obfuscations to determine the usability and scalability in producing correct sub trace matches. In particular, we are interested in dealing with variable renaming obfuscation via applying the trace slice to all possible data manipulators exist in a given execution trace. In this case, we may decrease the complexity problem of comparing semantic traces of malware variants without relying on variable names.

We are currently investigating how to develop a framework for determining the set of approximate semantic traces with respect to possible program execution paths in a program CFG. As a first step in this direction, we observe that for each (unique) execution path in a program CFG, there may exists a set of execution traces that might have similar semantics. Hence, an interesting research task consists in characterising the set of semantic abstractions which describe the relation between the abstract environment (i.e., approximate semantic traces) and the concrete environment (i.e., the control flow graph). This characterisation may be described as a Galois connection between two domains and may help us in reasoning about the minimisation of false negatives in matching trace slices.

References

[1] Flat assembler. http://flatassembler.net/.

[2] The netwide assembler. http://www.nasm.us/.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM, 1990.

[4] T. Akgul, V. J. M. III, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 522–531. IEEE Computer Society, 2004.

[5] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 184–189. IEEE Computer Society, 1999.

[6] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 188. IEEE Computer Society, 1997.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, jul 1997. URL http://www.cs.auckland.ac.nz/$\sim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[8] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[11] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, 2000.

[12] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *SCAM 2003: 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 118–. IEEE Computer Society, 2003.

[13] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[14] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[15] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 66–79. ACM, 1994.

[16] A. Majumdar, S. J. Drape, and C. D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 70–81. ACM, 2007.

[17] D. P. Mohapatra, R. Kumar, R. Mall, D. S. Kumar, and M. Bhasin. Distributed dynamic slicing of java programs. *J. Syst. Softw.*, 79(12):1661–1678, 2006.

[18] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper.*, 29(2):167–193, 1999.

[19] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM Press, 2007.

[20] J. Rilling, H. F. Li, and D. Goswami. Predicate-based dynamic slicing of message passing programs. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, page 133. IEEE Computer Society, 2002.

[21] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[22] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[23] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329. IEEE Computer Society, 2003.

[24] X. Zhang, R. Gupta, and Y. Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Trans. Program. Lang. Syst.*, 27(4):631–661, 2005.