Infections as Abstract Symbolic Finite Automata: Formal Model and Applications

Mila Dalla Preda*, Isabella Mastroeni†
University of Verona, Italy
*Email: mila.dallapreda@univr.it
†Email: isabella.mastroeni@univr.it

***Abstract*—In this paper, we propose a methodology, based on machine learning, for building a symbolic finite state automata-based model of infected systems, that expresses the interaction between the malware and the environment by combining in the same model the code and the semantics of a system and allowing to tune both the system and the malware code observation. Moreover, we show that this methodology may have several applications in the context of malware detection.**

Keywords: (Abstract) Symbolic finite state automata, Infection model, malware detection

I. Introduction

In these last decades we have witnessed at the raise of many different kinds of malware, targeting every day more, also mobile devices. The large amounts of malware sample are often mutations of known families, created either for avoiding detection or for dealing with different platforms. It is a common practice for malware to exploit both metamorphism and anti-emulation techniques to avoid respectively static and dynamic detection. Metamorphism refers to the use of obfuscation techniques to mislead automatic detection by camouflaging the syntax of malicious code. Anti-emulation techniques aim at detecting whether the malware is running in a protected environment in order to foil dynamic detection by hiding the malicious intent. This is only an example of how modern malware are often difficult to analyze due to their highly interactive nature that makes them exhibit the malicious behavior only when properly stimulated (this is a common practice in mobile malicious applications). Indeed, many malware include triggers that ensure that certain functions are invoked only when particular environmental or temporal conditions are satisfied. Common examples are bot that wait for inputs from their command and control servers, or logic bombs, which execute their malicious payload only before (or after) a certain event occurs. It is clear that this malware feature is a problem for detection schemas based on dynamic analysis, leading to false negatives (missed infections) due to the incomplete test coverage. It is therefore very important to study the interaction between the malware and the infected system (which can be either a desktop computer or a mobile device) in order to understand which are the events and actions of the system that properly stimulate the malicious behavior. On the other hand, it is also important to handle metamorphism, namely to use a behavioral model of malware that attempts to capture the essence of being malicious while abstracting from syntactic code details that may vary among different obfuscated variants of malware.

Hence, in order to defeat modern malware performing these kind of auto preservation techniques, we need a behavioral model that allows us to expresses the interaction between the system and the malware, and also to tune the level of precision of the syntax specification and of the code behavior of malware. Following this observation, we propose the use of abstract symbolic finite state automata [6] (SFA) for modeling malware behaviors. SFAs, introduced in [12] and further developed in [8], [7], provide the ideal formal setting for modeling, within the same framework, the abstraction of both the syntactic structure of programs and their intended semantics. Indeed, SFA have been introduced as an extension of traditional finite state automata for modeling languages with a potential infinite alphabet. Transitions in SFA are modeled as constraints interpreted in a given Boolean algebra, providing the semantic interpretation of constraints, and therefore the (potentially infinite) structural components of the language recognized (see [8], [12]). The notion of abstract SFA has been introduced in [6] where approximation is obtained by abstract interpretation. Abstract interpretation can act either at the syntactic (predicate), or at the topological (graph), or at the semantic (denotation) level. As presented in [6] SFAs model both the syntax and the semantics of systems. Indeed, following the structure of their control flow graphs, programs and systems can be represented as SFAs, where states are program points between basic blocks, the predicates labeling the transitions are the instructions of the basic blocks, and the interpretation of these predicates is given by the (possibly infinite) I/O semantics of each basic block. Hence, an SFA represents the code syntax and structure, while the language recognized by the SFA represents the semantics of the code fragment.

In order to build the SFA specification of infection dealing also with metamorphism, we assume to have access to a set of systems infected with different variants of a given malware. Then we follow an approach similar to machine learning for building a general SFA model for infection. Following the standard terminology of machine learning we call *training set* the set of known infected systems (where we suppose to know which actions are benign and which one are malicious). In order to be as proactive as possible, we do not consider simple infections, but families of variants of infections. This makes the detection process resilient, at least to the metamorphic variants analyzed in the training set. Moreover, we consider different kind of systems infected with variants of the same malware. In this way, we can observe many possible different ways in which a malware can interact with the infected system, namely in which the malicious behavior may be triggered. Next, we merge these samples of infected systems in order to construct a single SFA model of infection that expresses all the interactions between the malware and the systems considered in the training set. In machine learning, once a model is extracted from the training set it needs to be *generalized* in order to cover also cases that do not belong to the training set. Hence, the idea is to generalize the SFA model of infection by abstracting it wrt its syntax, approximating so far the specification of infection in order to recognize also infections not in the training set, but sharing the same syntactic *properties* of the training set witnesses. We may want to approximate either the benign actions, or the malicious actions, or both. By approximating benign actions we abstract actions of the infected system. In this way, we obtain a model of infection parametric on the abstraction of benign behaviors aiming at recognizing also infected systems that do not belong to the training set. At the limit, we can decide to lose all the information concerning benign actions by approximating these actions with $\top$ (representing any possible program fragment). In this case the model will highlight only *where* interactions between the malware and the system occur. On the other side, by generalizing malicious actions we approximate the malicious behaviors seen in the training set. The abstraction of the malicious behavior allows us to potentially recognize also metamorphic code variants not witnessed in the training set.

We believe that the abstract SFA model we propose have interesting applications in the malware detection scenario. This is an ongoing work, and therefore what we present here is simply a road-map of how we can build an SFA specifications of infection and of how the abstract SFA model could be used in malware detection.

The contributions of the paper are the following:

- We propose a methodology, based on machine learning, for building an SFA model of infected systems, that expresses the interaction between the malware and the environment, therefore potentially showing malware dormant behaviors. Moreover, this model allows us to tune the abstraction of both the benign and the malicious actions independently;
- We describe four possible applications of the proposed infection model to malware detection. Let us denote with $W_\eta[TS(\mathtt{M})]$ the abstract SFA model of infection extracted by machine learning on a training set $TS(\mathtt{M})$, of systems infected with variants of malware $\mathtt{M}$, and where η specifies a syntactic abstraction. Let $W[\mathtt{S}]$ be the SFA model of system $\mathtt{S}$ that we want to test for infection:
    a) Use $W_\eta[TS(\mathtt{M})]$ as a monitor for checking whether a particular execution contains infected behavior.
    b) $W_\eta[TS(\mathtt{M})]$ can be used as a signature for infection: a system $\mathtt{S}$ is infected with a malware $\mathtt{M}$ if there exists a behavior of $W_\eta[TS(\mathtt{M})]$ that appears in $W[\mathtt{S}]$.
    c) Extract from $W_\eta[TS(\mathtt{M})]$ a description of the malicious behaviors (also the dormant ones), and check whether the SFA $W[\mathtt{S}]$ exhibits these behaviors.
    d) Use $W_\eta[TS(\mathtt{M})]$ to drive the dynamic analysis of paths that seem to contain a dormant malicious behavior.

Structure of the paper: In Section II we briefly present abstract interpretation and the SFA model together with its syntactic abstraction. The section ends with the description of how SFA can be used to model systems. Next, in Section III we show how, from the SFA of an infected system in the training set, we can derive a model of infection, and how we can generalize it by abstracting either benign or malicious actions. In Section IV we discuss how the generalized model of infection can be used for malware detection.

II. PRELIMINARIES

Notation: Given two sets S and T, we denote with $\wp(S)$ the powerset of S, $\wp^{\mathrm{re}}(S)$ the set of recursive enumerable (r.e.) subsets of S, with $S \smallsetminus T$ the set-difference between S and T, with $S \subset T$ strict inclusion and with $S \subseteq T$ inclusion. S^* denotes the set of all finite sequences of elements in S. A set L with ordering relation $\leq$ is a poset and it is denoted as $\langle L, \leq\rangle$. A poset $\langle L, \leq\rangle$ is a lattice if $\forall x.y \in L$ we have that $x \vee y$ and $x \wedge y$ belong to L. A lattice $\langle L, \leq\rangle$ is complete when for every $X \subseteq L$ we have that $\bigvee X, \bigwedge X \in L$. As usual a complete lattice L, with ordering $\leq$, least upper bound (lub) $\vee$, greatest lower bound (glb) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\perp$ is denoted by $\langle L, \leq, \vee, \wedge, \top, \perp\rangle$.

Abstract Interpretation: Abstract domains can be formalized as sets of fix-points of closure operators on a given concrete domain which is a complete lattice C (cf. [5]). An *upper closure operator* (or simply a *closure*) on a complete lattice C is an operator $\rho : C \longrightarrow C$ which is monotone, idempotent, and extensive (i.e., $x \leq \rho(x)$). We denote with $uco(C)$ the set of all closure operators on the complete lattice C, namely it is the set of all possible abstractions of C. If C is a complete lattice, then $uco(C)$ forms a complete lattice [13] where the bottom is the identity closure $id = \lambda x.x$ and the top is the abstraction $\lambda x. \top$ unable to observe anything.

A. Symbolic Finite State Automata

Symbolic automata and finite state transducers have been introduced to deal with specifications involving a potentially infinite alphabet of symbols [8], [7], [12]. We follow [8] in specifying symbolic automata in terms of effective Boolean algebra. Consider an effective Boolean algebra $\mathcal{A} = \langle \mathfrak{D}_\mathcal{A}, \Psi_\mathcal{A}, [\![\cdot]\!], \perp, \top, \wedge, \vee, \neg\rangle$, with domain elements in a r.e. set $\mathfrak{D}_\mathcal{A}$ and predicates in a r.e. set $\Psi_\mathcal{A}$ closed under boolean connectives $\wedge$, $\vee$ and $\neg$. The semantic function $[\![\cdot]\!] : \Psi_\mathcal{A} \longrightarrow \wp^{\mathrm{re}}(\mathfrak{D}_\mathcal{A})$ is a partial recursive function such that $[\![\perp]\!] = \varnothing$, $[\![\top]\!] = \mathfrak{D}_\mathcal{A}$, and $\forall \varphi, \phi \in \Psi_\mathcal{A}$ we have that $\llbracket \varphi \vee \phi \rrbracket = \llbracket \varphi \rrbracket \cup \llbracket \phi \rrbracket$, $\llbracket \varphi \wedge \phi \rrbracket = \llbracket \varphi \rrbracket \cap \llbracket \phi \rrbracket$, and $\llbracket \neg\varphi \rrbracket = \mathfrak{D}_{\mathcal{A}} \smallsetminus \llbracket \varphi \rrbracket$. In the following, we abuse notation by denoting with $\llbracket\cdot\rrbracket$ also its additive lift to $\wp(\Psi_{\mathcal{A}})$, i.e., for any $\Phi \in \wp(\Psi_{\mathcal{A}})$: $\llbracket \Phi \rrbracket = \{ \llbracket \varphi \rrbracket \mid \varphi \in \Phi \}$. For $\varphi \in \Psi_{\mathcal{A}}$ we write $\texttt{IsSat}(\varphi)$ when $\llbracket \varphi \rrbracket \neq \varnothing$ and we say that φ is *satisfiable*. $\mathcal{A}$ is decidable if $\texttt{IsSat}$ is decidable.

Definition 1: A symbolic finite automaton (SFA) W is a tuple $\langle \mathcal{A}, Q, q_0, F, \Delta \rangle$ where $\mathcal{A}$ is an effective Boolean algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of transitions.

A transition in $W = \langle \mathcal{A}, Q, q_0, F, \Delta \rangle$, labeled φ from state p to state q, $(p, \varphi, q) \in \Delta$, is also denoted $p \xrightarrow{\varphi} q$. φ is called the *guard* of the transition. An a-move of an SFA W is a transition $p \xrightarrow{\varphi} q$ such that $a \in \llbracket \varphi \rrbracket$, also denoted $p \xrightarrow{a} q$. The language of a state $q \in Q$ in W is defined as:

$$\mathscr{L}_q(W) = \left\{ a_1, \dots, a_n \in \mathfrak{D}_{\mathcal{A}} \;\middle|\; \begin{array}{l} \forall 1 \leq i \leq n.\ p_{i-1} \xrightarrow{a_i} p_i \\ p_0 = q,\ p_n \in F \end{array} \right\}$$

in this case, $\mathscr{L}(W) = \mathscr{L}_{q_0}(W)$. We assume *complete SFA*, namely where all states hold an out-going a-move, for any character $a \in \mathfrak{D}$. This can be simply achieved by adding a shaft-state $q_\perp \in Q$ such that $q_\perp \xrightarrow{\top} q_\perp \in \Delta$ and for all states q lacking an out-going a-move, for $a \in \mathfrak{D}$, then $q \xrightarrow{\neg\beta} q_\perp \in \Delta$ with $\beta = \bigvee \{ \varphi \mid q \xrightarrow{\varphi} p \ \wedge\ p \in Q \}$.

Given an SFA $W = \langle \mathcal{A}, Q, q_0, F, \Delta \rangle$ and $\equiv \subseteq Q \times Q$, we define the *quotient SFA* $W_{/\equiv} \overset{\text{def}}{=} \langle \mathcal{A}, Q', q_0', F', \Delta' \rangle$ as follows: $Q' = \{ [q]_\equiv \mid q \in Q \}$, $\Delta' \subseteq Q' \times \Psi_{\mathcal{A}} \times Q'$ is such that $\Delta' = \{ ([q]_\equiv, \Phi, [q']_\equiv) \mid (p, \Phi, q') \in \Delta, p \in [q]_\equiv \}$, $q_0' = [q_0]_\equiv$, and $F' = \{ [q]_\equiv \mid q \in F \}$.

In [9] the authors extend some classical constructs, such as language intersection and language containment, to SFA, while in [8] the authors present an efficient minimization algorithm for SFA.

We define the projection of a language $\mathscr{L}(W) \subseteq \Sigma^*$ with respect to a subset $X \subseteq \Sigma$ of the alphabet as follows: $\mathit{Proj}(\mathscr{L}(W), X) = \{\pi(s, X) \mid s \in \mathscr{L}\}$, where given $s = s_1 s_2 \dots s_n$ the string projection function π is defined as follows

$$\begin{cases} \pi(\epsilon, X) = \epsilon \\ \pi(s_1 s_2 \dots s_n, X) = \begin{cases} s_1 \pi(s_2 \dots s_n, X) & \text{if } s_1 \in X \\ \pi(s_2 \dots s_n, X) & \text{otherwise} \end{cases} \end{cases}$$

B. Abstract Symbolic Finite Automata

Recently, it has been developed a general framework for abstracting SFA [6]. In particular, an SFA can be abstracted either by approximating the domain of predicates labeling the edges, or by approximating the domain of denotations expressing the meaning of predicates. In both cases we enlarge the language recognized by the SFA. In [6] there is a rigorous description of both syntactic and semantic abstraction of SFA and of their strong relation. In this work, we are interested mainly in the SFA syntactic abstraction and therefore formally present only this SFA abstraction. The syntactic abstraction of SFA is an abstract interpretation-based approximation of the domain of predicates $\wp^{\mathtt{re}}(\Psi_{\mathcal{A}})$ of the underlying effective Boolean algebra $\mathcal{A}$.

Definition 2: Let $\mathcal{A}$ be an effective Boolean Algebra and let $\eta \in uco(\wp^{\mathtt{re}}(\Psi_{\mathcal{A}}))$ be an additive abstraction of predicates. The syntactic abstraction of $\mathcal{A}$ w.r.t. η, denoted $\langle\eta\rangle$-abstraction, is the effective Boolean algebra

$$\mathcal{A}_\eta = \langle \mathfrak{D}_{\mathcal{A}}, \eta(\wp^{\mathtt{re}}(\Psi_{\mathcal{A}})), \llbracket\cdot\rrbracket, \perp, \top, \wedge, \vee, \neg \rangle$$

where $\llbracket\cdot\rrbracket : \eta(\wp^{\mathtt{re}}(\Psi_{\mathcal{A}})) \longrightarrow \wp(\mathfrak{D}_{\mathcal{A}})$ is defined as in SFA.

Consider an SFA $W = \langle \mathcal{A}, Q, q_0, F, \Delta \rangle$ and the $\langle\eta\rangle$-abstraction of the effective Boolean algebra $\mathcal{A}$, denoted as $\mathcal{A}_\eta$. We define the symbolic finite automaton corresponding to W on the abstract effective Boolean algebra $\mathcal{A}_\eta$ as $W_\eta \overset{\text{def}}{=} \langle \mathcal{A}_\eta, Q, q_0, F, \Delta_\eta \rangle$ where:

$$\Delta_\eta \overset{\text{def}}{=} \{ (q, \eta(\varphi), q') \mid (q, \varphi, q') \in \Delta \}$$

In the following we abuse terminology by calling $\langle\eta\rangle$-abstract SFA, denoted W_η, the SFA whose underlying Boolean algebra is an $\langle\eta\rangle$-abstraction of a Boolean algebra $\mathcal{A}$. It is possible to prove that, when abstracting the underling effective Boolean algebra of an SFA, we over-approximate the recognized language, namely $\mathscr{L}(W) \subseteq \mathscr{L}(W_\eta)$ [6].

C. Systems as SFA

According to [6], in this section we specify the approximate semantics of a system as the language recognized by an SFA. We consider systems in imperative computational model and assume to have access their correct control flow graph (CFG). The CFG of a system is a graph whose nodes are sequences of non-branching instructions. More formally, let $\mathbf{I}$ be the instruction set containing both branching and non-branching instructions. We denote by $\mathbb{I} \subseteq \mathbf{I}$ the set of non-branching instructions and by $\mathbb{C}$ the set of boolean expressions over system states, namely the set of guards in branching instructions. Let $\mathtt{c}$ range over $\mathbb{C}$ and $\mathtt{b}$ range over $\mathbb{I}^*$. The CFG of a system $\mathtt{S}$ is a graph $G_{\mathtt{S}} = (N_{\mathtt{S}}, E_{\mathtt{S}})$ where the set $N_{\mathtt{S}} \subseteq \mathbb{I}^*$ of nodes specifies the basic blocks of $\mathtt{S}$, namely the maximal sequences of sequential instructions of $\mathtt{S}$, while the set of edges $E_{\mathtt{S}} \subseteq N_{\mathtt{S}} \times \mathbb{C} \times N_{\mathtt{S}}$ denotes the guarded transitions of $\mathtt{S}$. In particular, a labeled edge $(\mathtt{b}, \mathtt{c}, \mathtt{b}') \in E_{\mathtt{S}}$ means that the execution of $\mathtt{S}$ flows from $\mathtt{b}$ to $\mathtt{b}'$ when the execution of $\mathtt{b}$ leads to a program state that satisfies condition $\mathtt{c}$. When a basic block $\mathtt{b}$ has no outgoing edges in $E_{\mathtt{S}}$ we say that it is final, denoted $\mathtt{b} \in \mathit{Final}[G_{\mathtt{S}}]$. We denote by $\mathit{in}[\mathtt{b}]$ and $\mathit{out}[\mathtt{b}]$ respectively the entry and exit point of the basic block $\mathtt{b}$, and with $\mathbb{PP}[G_{\mathtt{S}}]$ the block delimiters of $G_{\mathtt{S}}$, namely the set of all the entry and exit points of the basic blocks of $G_{\mathtt{S}}$. Formally:

$$\mathbb{PP}[G_{\mathtt{S}}] \overset{\text{def}}{=} \{ \mathit{in}[\mathtt{b}] \mid \mathtt{b} \in N_{\mathtt{S}} \} \cup \{ \mathit{out}[\mathtt{b}] \mid \mathtt{b} \in N_{\mathtt{S}} \}$$

Let Σ, ranged over by s, be the set of possible system states. Let $\mathit{exec} : \mathbb{I}^* \longrightarrow \wp^{\mathtt{re}}(\Sigma \times \Sigma)$ be the function that defines the semantics of basic blocks, namely the pairs of input/output states that model the execution of the sequences of instructions in a block. When $(s, s') \in \mathit{exec}(\mathtt{b})$ it means that the execution of the sequence of instructions $\mathtt{b}$ transforms state s into state

s'. Let us denote by $s \models \mathtt{c}$ the fact that the boolean condition $\mathtt{c}$ is satisfied by state $s \in \Sigma$.

We define the set of executions of the CFG of a system $\mathtt{S}$ as the set of all the sequences of basic blocks and guards that can be encountered along a path of $G_{\mathtt{S}} = (N_{\mathtt{S}}, E_{\mathtt{S}})$. Formally:

$$Exe[G_{\mathtt{S}}] \overset{\text{def}}{=} \left\{ \mathtt{b}_0\mathtt{c}_1\mathtt{b}_1\mathtt{c}_2 \dots \mathtt{c}_k\mathtt{b}_k \,\middle|\, \begin{array}{l} \forall 0 \leq i < k : \\ (\mathtt{b}_i, \mathtt{c}_{i+1}, \mathtt{b}_{i+1}) \in E_{\mathtt{S}} \end{array} \right\} \quad (1)$$

The execution trace semantics of a system $\mathtt{S}$, denoted $[\![\mathtt{S}]\!]$, is therefore the set of all finite executions [10] starting from the entry point of the starting basic block $\mathtt{b}_0$ in the CFG $G_{\mathtt{S}}$ of $\mathtt{S}$. Let $Init_{\mathtt{S}} \subseteq \Sigma$ be the set of possible initial states of system $\mathtt{S}$. Formally, for each $s_0 \in Init_{\mathtt{S}}$:

$$\begin{aligned} [\![\mathtt{S}]\!](s_0) \overset{\text{def}}{=} \{ & (s_0, s_1)(s_1, s_1)(s_1, s_2) \dots (s_k, s_k)(s_k, s_{k+1}) \mid \\ & \mathtt{b}_0\mathtt{c}_1\mathtt{b}_1 \dots \mathtt{c}_k\mathtt{b}_k \in Exe[G_{\mathtt{S}}], \\ & \forall 0 < i \leq k : s_i \models \mathtt{c}_i, (s_{i-1}, s_i) \in exec(\mathtt{b}_{i-1}) \} \end{aligned}$$

$$[\![\mathtt{S}]\!] \overset{\text{def}}{=} \bigcup \left\{ [\![\mathtt{S}]\!](s_0) \,\middle|\, s_0 \in Init_{\mathtt{S}} \right\}$$

In order to define the SFA that corresponds to the CFG semantics of a given system we need to define an effective Boolean algebra that it is suitable for the representation of system execution. For this reason we define the following effective Boolean algebra where predicates are either basic blocks of instructions or guards of branching instructions, representing the syntactic structure of the system, and the denotations are pairs of input/output states:

$$\mathfrak{P} \overset{\text{def}}{=} \langle \Sigma \times \Sigma, \mathbb{I}^* \cup \mathbb{C}, \{\!| \cdot |\!\}, \bot, \top, \wedge, \vee, \neg \rangle$$

where the semantic function $\{\!| \cdot |\!\} : \mathbb{I}^* \cup \mathbb{C} \longrightarrow \wp^{\mathtt{re}}(\Sigma \times \Sigma)$ is defined as follows for $\varphi \in \mathbb{I}^* \cup \mathbb{C}$:

$$\{\!|\varphi|\!\} \overset{\text{def}}{=} \left\{ \begin{array}{ll} \left\{ (s, s') \,\middle|\, (s, s') \in exec(\mathtt{b}) \right\} & \text{if } \varphi = \mathtt{b} \in \mathbb{I}^* \\ \left\{ (s, s) \,\middle|\, s \models \mathtt{c} \right\} & \text{if } \varphi = \mathtt{c} \in \mathbb{C} \end{array} \right.$$

we denote by $\{\!| \cdot |\!\}$ also its point-wise extension to $\wp^{\mathtt{re}}(\mathbb{I}^* \cup \mathbb{C})$.

Definition 3: Let $\mathtt{S}$ *be a system with CFG* $G_{\mathtt{S}}$*. The SFA associated with* $\mathtt{S}$ *is*

$$W[\mathtt{S}] \overset{def}{=} \langle \mathfrak{P}, \mathbb{PP}[G_{\mathtt{S}}], in[b_0], \{out[b] \mid b \in Final[G_{\mathtt{S}}]\}, \Delta_{\mathtt{S}} \rangle$$

where b_0 *is the starting basic block of* $G_{\mathtt{S}}$ *and* $\Delta_{\mathtt{S}}$ *is defined as:*

$$\begin{aligned} \Delta_{\mathtt{S}} \overset{def}{=} & \left\{ (in[b], b, out[b]) \,\middle|\, b \in N_{\mathtt{S}} \right\} \cup \\ & \left\{ (out[b], c, in[b']) \,\middle|\, (b, c, b') \in E_{\mathtt{S}} \right\} \end{aligned}$$

It has been proved in [6] that the language $\mathscr{L}(W[\mathtt{S}]) \in \wp^{\mathtt{re}}((\Sigma \times \Sigma)^*)$ recognized by the SFA $W[\mathtt{S}]$ approximates the concrete system semantics $[\![\mathtt{S}]\!]$ in a language of sequences of infinitely many possible input/output relations associated with each basic block, namely if $\mathtt{S}$ is a system then for any $s_0 \in Init_{\mathtt{S}}$: $[\![\mathtt{S}]\!](s_0) \in \mathscr{L}(W[\mathtt{S}])$.

Observe that the language recognized by $W[\mathtt{S}]$ is given by the strings consisting in the input/output pairs associated to the semantics of the instructions along the paths of the CFG of $\mathtt{S}$. Let us restrict the language to the effective computations, namely to those strings where the output state of a denotation is equal to the input of the next one. Formally, we define $\widehat{\mathfrak{L}}(W[\mathtt{S}])$ as follows:

$$\widehat{\mathfrak{L}}(W[\mathtt{S}]) \overset{\text{def}}{=} \left\{ \langle s_0^{in}, s_0^{out} \rangle \dots \langle s_k^{in}, s_k^{out} \rangle \,\middle|\, \forall i \in [0, k-1], s_i^{out} = s_{i+1}^{in} \right\}$$

Then, the following relation between the semantics of $\mathtt{S}$ and the set of effective computations recognized by $W[\mathtt{S}]$ holds.

Thorem 1: $\widehat{\mathfrak{L}}(W[\mathtt{S}]) = [\![\mathtt{S}]\!]$.

III. Learning infections

Consider an integration function $\mathfrak{I} : Systems \times Systems \to Systems$ such that $\mathfrak{I}(\mathtt{M}, \mathtt{S})$ denotes system $\mathtt{S}$ infected with malware $\mathtt{M}$, namely the function $\mathfrak{I}$ models the infection procedure. For instance, the function $\mathfrak{I}$ may trivially be the appending of a malware to an executable file. The training set we consider is a collection of infected systems, and for each infected system we assume to know which instructions are of the malware and which are of the original/benign system. We consider the training set for malware $\mathtt{M}$ as given by:

$$TS(\mathtt{M}) = \{\mathfrak{I}(\mathtt{M}_1, \mathtt{S}_1) \dots \mathfrak{I}(\mathtt{M}_k, \mathtt{S}_k)\}$$

where $\mathtt{M}_i$ are metamorphic variants of $\mathtt{M}$ and $\mathtt{S}_i$ are benign programs. In the following, when $\mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i) \in TS(\mathtt{M})$ we may write $\mathtt{S}_i \in TS(\mathtt{M})$ and $\mathtt{M}_i \in TS(\mathtt{M})$. Given $TS(\mathtt{M})$ we construct the SFA of each infected system:

$$W[\mathfrak{I}(\mathtt{M}_1, \mathtt{S}_1)] \dots W[\mathfrak{I}(\mathtt{M}_k, \mathtt{S}_k)]$$

Then we compute the SFA that embeds all the infections seen in the training set:

$$W[TS(\mathtt{M})] \overset{\text{def}}{=} \uplus \left\{ W[\mathfrak{I}(\mathtt{M}_i, \mathtt{S}_1)] \,\middle|\, \mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i) \in TS(\mathtt{M}) \right\}$$

where $\uplus$ is the minimization [8] of the union of the SFA in the considered set. Thus, the language of real computations recognized by $W_{TS(\mathtt{M})}$ corresponds to the computations of all the infected systems seen in the training set.

Proposition 1: Given a training set $TS(\mathtt{M})$ *we have that:*

$$\widehat{\mathfrak{L}}(W[TS(\mathtt{M})]) = \bigcup \left\{ [\![\mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i)]\!] \,\middle|\, \mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i) \in TS(\mathtt{M}) \right\}$$

Proof. By definition of $\widehat{\mathfrak{L}}$ we have that $\widehat{\mathfrak{L}}(W_{TS(\mathtt{M})}) = \bigcup \left\{ \widehat{\mathfrak{L}}(W[\mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i)]) \,\middle|\, \mathfrak{I}(\mathtt{M}_i, \mathtt{S}_i) \in TS(\mathtt{M}) \right\}$, therefore the thesis follows from Theorem 1. □

Generalization

The idea of the generalization process consists in starting from an SFA $W[TS(\mathtt{M})]$ and abstracting either the benign or the malicious actions of the resulting system. In order to be able to abstract independently the benign and malicious actions we need to augment the domain of predicates with a flag expressing whether the considered action is malicious or not. From now on, we consider the following Boolean algebra for representing programs:

$$\mathfrak{P}^+ \overset{\text{def}}{=} \langle \Sigma \times \Sigma, \{\bullet, \circ\} \times (\mathbb{I}^* \cup \mathbb{C}), \{\!| \cdot |\!\}, \bot, \top, \wedge, \vee, \neg \rangle$$

where a predicate is now a pair, where for example the predicate $(\circ, \mathtt{b})$ denotes a benign sequence of instructions, while a predicate $(\bullet, \mathtt{b})$ a malicious sequence of instructions. Let us define the following abstractions:

- $\eta^\bullet \in uco(\wp(\{\bullet\} \times (\mathbb{I}^* \cup \mathbb{C})))$ that specifies how malicious actions are abstracted
- $\eta^\circ \in uco(\wp(\{\circ\} \times (\mathbb{I}^* \cup \mathbb{C})))$ that specifies how benign actions are abstracted

The abstraction $\eta \in uco(\wp(\{\bullet, \circ\} \times (\mathbb{I}^* \cup \mathbb{C})))$ of flagged predicates can then be specified as the combination of the malicious and benign abstraction: $\eta = \eta^\bullet \times \eta^\circ$. We denote by $W_\eta[TS(\mathtt{M})]$ the SFA $W[TS(\mathtt{M})]$ abstracted by the syntactic abstraction η. Observe that, by definition of SFA syntactic abstraction, we have that $\mathscr{L}(W[TS(\mathtt{M})]) \subseteq \mathscr{L}(W_\eta[TS(\mathtt{M})])$, which means that the abstract model we build recognizes at least all the behaviors in the training set. In particular, the behaviors recognized by the abstract SFA $\mathscr{L}(W_\eta[TS(\mathtt{M})])$ are precisely those expressing the generalization operated by the abstraction.

Note that, the abstraction η is defined in such a way that we can *separately tune* the abstraction of benign and of malicious actions. For example, an abstraction $\eta = id \times \eta^\circ$ denotes an abstraction of predicates that generalizes/abstracts only benign actions, while an abstraction $\eta = \eta^\bullet \times id$ represents the generalization/abstraction of only malicious actions.

For example, if we want to lose all the information concerning benign actions, then we can use the abstraction $\eta = id \times \lambda x.\ \top$ abstracting all the benign behaviors to $\top$, denoting the possibility of executing any possible benign computation. This leads to a behavioral model of infection where $\mathtt{M}$ is precisely represented in the computation model, while we lose any information regarding the benign actions seen in the training set. The SFA obtained so far specifies only when some (possibly any) interaction with the benign system need to occur. It is worth noting that this kind of abstraction could be used for highlighting the kind of infection used by the malware $\mathtt{M}$. For example, simple prepending or appending viruses will have an abstract SFA where the $\top$ abstract predicates are all, respectively, at the beginning or at the end of the SFA. We can imagine that the SFA of system infected with logic bombs, or malware with dormant behaviors will have a branch that exhibits non-$\top$ predicates only in one branch. The idea is that, by abstracting the benign actions on $\top$ the topology of the SFA could reveal something regarding the kind of infection.

On the other hand, if we are interested in detecting the possible variants of malware that can infect a system considered in the training set, we can use an abstraction $\eta = \eta^\bullet \times id$, where abstraction $\eta^\bullet$ models the information that we want to lose on the malicious variants seen in the training set, in order to become able to recognize other metamorphic variants of the same malware. Standard research on behavioral malware detection has focused on the definition of these kind of abstractions of malicious actions. Interestingly, the proposed approach would allow us to *combine* the specification of a metamorphic malware behavior with the interactions with the surrounding system in the same model of infection.

IV. Applications to Malware Detection: Ideas and Future Directions

In this section, we discuss some possible applications of the proposed model of infection $W_\eta[TS(\mathtt{M})]$ in the context of malware detection. These possible applications are briefly described deserving future work.

A. *Dynamic Detection*

The infection model $W_\eta[TS(\mathtt{M})]$ could be used to guide a monitor algorithm that checks if a particular execution matches an infected behavior expressed by $W_\eta[TS(\mathtt{M})]$. Thus, given an input that generates an execution trace w the monitor algorithm dynamically checks if:

$$w \in \widehat{\mathcal{L}}(W_\eta[TS(\mathtt{M})])$$

Consider the following notation: if $w = (s_0\, s_1\, s_2 \dots s_n) \in \Sigma^*$ then $hd(w) = s_0 \in \Sigma$ while $tl(w) = s_1\, s_2 \dots s_n \in \Sigma^*$, and $\Delta^q \overset{\text{def}}{=} \left\{\ (q, \varphi, p) \,\middle|\, (q, \varphi, p) \in \Delta\ \right\}$. In Fig. 1 we present the dynamic detection algorithm monitoring whether an execution trace shows malicious, or at least suspicious, behaviors. The idea of the algorithm consists in receiving in input a trace of execution states. For each state in this sequence we check which paths can be followed. We follow an edge whenever we are in a system state belonging to the semantics of, at least one, predicate labeling the edge. Moreover, if the satisfied predicate is concrete (i.e., is the only possibility) then we erase the edge from the set of the further feasible paths, otherwise (when the the label is an abstract predicate consisting in a set of predicates) we keep the edge as feasible while keeping trace of the reached automaton states.
Let us show this idea on an example. Consider the SFA in Fig. 2 with final state q_6. Let us suppose that $\varphi_1 = 0 \le x < 5$, $\varphi_2 = 5 \le x < 10$ and $\varphi_3 = 10 \le x < 15$. The set of predicates are abstractions: $\eta = id \times \eta^\circ$ where $\eta^\circ(\varphi_1) = \eta^\circ(\varphi_2) = \{\varphi_1, \varphi_2, \varphi_3\}$, $\eta^\circ(\varphi_3) = \{\varphi_2, \varphi_3\}$. The idea is that, if an execution trace reaches the final state it means that it contains a sequence of malicious actions (the non-abstracted predicates) which may correspond to a suspicious execution, and then it is stopped with an alarm. In the picture, we use the algorithm for verifying whether w_a and/or w_b (sequences of values for x) reach q_6 in the depicted SFA. In particular, starting from the frontier (of reachable states) denoted by dashed closed lines, we end in the ones denoted with plain closed lines. We can observe that in the case (a) the trace allows to reach the final state q_6, hence we stop the monitor raising an alarm, meaning that a potential malicious behavior is recognized in our system. In the case (b) the last frontiers contains only the state q_3 which is not final, hence we have not recognized any malicious complete behavior.

B. $W_\eta[TS(\mathtt{M})]$ *as signature of infection*

By definition, $W_\eta[TS(\mathtt{M})]$ is the generalized model of infection of a malware $\mathtt{M}$ extracted from the training set

```
**DDetection**$(W, w)$
input: $W = (\mathcal{A}, Q, q_0, F, \Delta)$, $\eta \in uco(\wp^{\mathrm{re}}(\Psi_{\mathcal{A}}))$
    $\mathcal{A} = (\mathcal{D}_{\mathcal{A}}, \Psi_{\mathcal{A}}, \{\!|\cdot|\!\}, \bot, \top, \vee, \wedge, \neg)$, $w \in \Psi_{\mathcal{A}}^*$
Fr $= \Delta_\eta^{q_0}$; FrT = Fr; $\mathtt{R} = \{q_0\}$; output=wait;
**while** $w \neq$ **null** $\wedge$ output==wait **do**
    {**while** FrT $\neq$ **null** $\wedge$ output==wait **do**
        {select and remove $(q, \Phi, p)$ from FrT
        Fr = Fr $\smallsetminus \{(q, \Phi, p)\}$;
        $\mathtt{R} = \mathtt{R} \smallsetminus \{q\}$;
        **if** $(\exists \varphi \in \Phi.\, hd(w) \in [\![\varphi]\!])$ **then**
            {Fr = Fr $\cup \Delta_\eta^p$; $\mathtt{R} = \mathtt{R} \cup \{p\}$;
            **if** $(|\Phi| > 1)$ **then**
                {Fr = Fr $\cup \{(p, \Phi, p)\}$; }}
        **if** $(\mathtt{R} \cap F \neq \varnothing)$ **then** {output=alarm; }
    **if** $(w ==$ **null**$)$ **then** {output=ok; }
    FrT = Fr; $w = tl(w)$; }}
**return** output;
```

Fig. 1. Dynamic detection algorithm

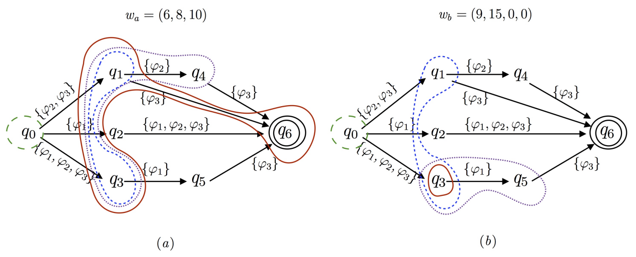

Fig. 2. Example of dynamic detection.

$TS(\mathtt{M})$. This abstract model of infection can be used as a signature for the analysis of systems that are infected by $\mathtt{M}$. More formally, we say that a system $\mathtt{S}$ is infected with $\mathtt{M}$ if

$$\widehat{\mathcal{L}}(W_\eta[TS(\mathtt{M})]) \cap \widehat{\mathcal{L}}(W[\mathtt{S}]) \neq \emptyset \tag{2}$$

namely if there exists a behavior of the abstract infection model that appears also in $\mathtt{S}$.

Note that, the semantics of predicates are r.e. sets, hence to decide whether a sequence belongs to the language of an SFA is a semi-decidable problem, in other words the language of an SFA is a r.e. set. This means that also the decision problem (2) is semi-decidable: we use the algorithm for semi-deciding whether a r.e. set is empty and each time we find a sequence in the language, in parallel, we test if the sequence belongs also to the other language. It is worth noting that it is well known that the to decide whether the intersection of *regular* languages is the emptyset is decidable. We believe that it is possible to provide an algorithm for deciding problem (2) by adapting the decision algorithm for regular languages to the SFA language of predicates, being the set of predicates finite when dealing with CFG of systems. Indeed, finite state automata can be easily transformed into an SFA by considering as semantics precisely the predicate.

The idea of representing malicious behavior as languages recognized by automata is not new, see for example [1], [3], [4]. To the best of our knowledge, existing works modeling malicious behaviors by means of automata typically use a specific abstract language attempting to generalize and to capture metamorphic variants. Such abstract languages express either syntactic properties of the code (e.g., use of symbolic names to handle renaming) or execution properties extracted through static analysis. The main advantage in using the abstract SFA model instead of standard automata comes from the fact that when using SFA to model systems (see Section II-C) we have an unique model that incorporates both the syntax and the semantics of code thus allowing us to model both how the code is written and what it computes. Indeed, as presented in [6], abstract SFAs allow us to approximate both the syntax and the semantics of the predicates. Moreover, by abstracting the benign actions we may be able to identify infections of systems not considered in the training set, while by abstracting the malicious actions we may be able to recognize possible metamorphic variants of $\mathtt{M}$ not considered in the training set.

C. *Extract metamorphic malware signature*

Another possible exploitation of the proposed model is based on the idea of projecting the language recognized by $W_\eta[TS(\mathtt{M})]$ only on the malicious actions. In this way, we are extracting a malware model which may potentially include also behaviors whose execution depends on some external action (usually performed by the infected system).

Let $\mathscr{L}_\eta^\circ(TS(\mathtt{M}))$ be the language of strings obtained by projecting $\widehat{\mathcal{L}}(W_\eta[TS(\mathtt{M})])$ on malicious actions, namely $\mathscr{L}_\eta^\circ(TS(\mathtt{M})) = Proj(\widehat{\mathcal{L}}(W_\eta[TS(\mathtt{M})]), \{\circ\} \times \mathbb{I}^* \cup \mathbb{C})$. In this context, a detector could test if a system $\mathtt{S}$ is infected with $\mathtt{M}$ by checking if there exists a malicious string in $\mathscr{L}_\eta^\circ(TS(\mathtt{M}))$ that is a sub-string (potentially not sequential) of a string of the language recognized by the SFA $W[\mathtt{S}]$ modeling the system. The idea of considering potentially not sequential sub-string comes from the fact that between two malicious actions the infected system could perform any possible benign action.

Definition 4: [Not sequential sub-string] Let Σ be an alphabet. Let $u, v \in \Sigma^*$ such that $|u| \leq |v|$. u is a not sequential sub-string of v if there exists $\{u_i\}_{i\in[0,n]}, \{v_i\}_{i\in[0,m]} \in \wp(\Sigma^*)$ such that $u = u_0u_1 \ldots u_n$, $v = v_0v_1 \ldots v_m$, $m \geq m$, and $\forall i \in [0, n]\, \exists j \in [0, m].\, u_i = v_j \;\wedge\; j \geq i$.

Then we denote by $NSSstring(\mathscr{L}(W[\mathtt{S}]))$ the set of all the not sequential sub-strings of words in the language of $\mathscr{L}(W[\mathtt{S}])$. Thus, the infection test is:

$$\mathscr{L}_\eta^\circ(TS(\mathtt{M})) \cap NSSstrings(\mathscr{L}(W[\mathtt{S}])) \neq \emptyset \tag{3}$$

Note that, the language $\mathscr{L}_\eta^\circ(TS(\mathtt{M}))$ expresses only the generalization of the malicious behavior according to η. Such a generalization of the malicious behavior allows the detector to recognize possible metamorphic variants of the malware $\mathtt{M}$ sharing the same abstraction of $\mathtt{M}$ but not considered in the training set $TS(\mathtt{M})$.

As before, we can observe that the decision problem (3) is semi-decidable, but again we believe that it is possible to transform this decision problem in a decision problem on regular languages where again this problem become decidable. In particular, for finite state automata, given an automaton A whose language is denoted by $\mathcal{L}(A)$, the language

$\mathit{NSSstrings}(\mathcal{L}(A))$ is still regular. The intuition is that we can transform the problem of deciding whether w is a not sequential sub-string of $w' \in \mathcal{L}(A)$ building the automaton (with ϵ transitions, denoting moves possible without reading symbols) recognizing all not sequential sub-strings of $\mathcal{L}(A)$.

Definition 5: [NSSstring automaton] Let $A = \langle q_0, Q, F, \delta, \Sigma\rangle$ *be the finite state automaton with set of state Q, initial state* $q_0 \in Q$*, final states* $F \subseteq Q$*, alphabet* Σ *and transition relation* $\delta : Q \times \Sigma \longrightarrow Q$*. The corresponding NSSstring non deterministic automaton with* ϵ*-transitions is* $NSSs(A) = \langle q_0, Q, F, \delta', \Sigma\rangle$ *where* $\delta' : Q \times \Sigma \cup \{\epsilon\} \longrightarrow \wp(Q)$ *is defined as* $\delta' = \delta \cup \left\{ (q, \epsilon) \mapsto q' \;\middle|\; \exists a \in \Sigma.\, \delta(q, a) = q' \right\}$.
We observed in the previous section that a finite state automaton can be easily transformed into an SFA, moreover if the finite state automaton contains ϵ-transitions then the SFA corresponding edges are labeled with $\top$, meaning that any system state can be considered to follow that edge. Hence again, as before, in order to make decidable the decision problem (3) we should have to transform it as a decision problem on the language of predicates.

D. Exploring Dormant Behaviors

When using dynamic analysis for testing a system $\mathtt{S}$ for infection the main problem is the code coverage. Namely, dynamic analysis tests a particular execution of the system for infection and if the considered execution does not exhibit the malicious behavior we cannot be sure that another execution would not exhibit it. This is particularly true for modern malware that often exhibit malicious behaviors only when properly stimulated. For example, there are malware that use anti-emulation or anti-analysis techniques, namely that interrogate the environment in order to understand whether it is running in a virtual machine and, in this case, modify their behavior to make analysis harder. There are malware that perform the malicious intent only when certain conditions are met. Other typical examples are the bots that wait to be triggered by precise sequences of messages sent from the botnet controller. Thus, it is a common practice, in modern malware, to have dormant behaviors. We call triggering predicates those control points that have only one branch that exhibit a malicious behavior. To overcome this problem researchers have proposed techniques, as for example [2], [11], for exploring multiple execution paths during dynamic analysis in order to catch such dormant behaviors. These techniques often relay on the use of symbolic execution in order to find inputs that would lead to an execution that explores different paths. In order to focus on the execution of dormant behaviors these techniques should force the execution of both branches controlled by triggering predicates. This is done by marking all the system predicates that control conditional branches depending on external conditions, namely branching that may depend on malicious tests. For example, in [2], [11] all the predicates that depend on the answer received by system calls (as for example, current time of the operating system, battery consumption, content of a file, check for Internet connectivity) are marked and inputs are generated in order to execute both branches controlled by such predicates. The idea is to capture in this way the points of the system where control flow decisions may hide dormant behaviors. Of course this method, used for identifying triggering predicates, suffers from both false positives and false negatives. We believe that the abstract SFA model of infection $W_\eta[TS(\mathtt{M})]$ could be use to mark as triggering predicates those control points that have the characteristic that one branch exhibit a malicious behavior while the other does not. Indeed, this graphical property of control points identifies where the execution performs some kind of check and then decide whether to perform the malicious actions or not. Thus, it looks like a reasonable approximation of what we called triggering predicates. The idea would be to re-implement existing techniques, such as the ones in [2], [11], by using the SFA structure and labeling for identifying these control points where we force dynamic analysis to considers both branches. It would then be interesting to compare the dormant behaviors discovered in this way with the ones discovered in [2], [11].

Acknowledgments: This work is partly supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT.

References

[1] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification - First International Conference, RV10*, volume 6418 of *LNCS*, pages 168–182, London, UK, 2010. Springer-Verlag.

[2] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection: Countering the Largest Security Threat*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.

[3] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, pages 169–186, 2003.

[4] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, 2005.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282, New York, 1979. ACM Press.

[6] M. Dalla Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 329–341. ACM.

[7] L. D'Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 624–639. Springer, 2013.

[8] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 541–554. ACM, 2014.

[9] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.

[10] I. Mastroeni and R. Giacobazzi. An abstract interpretation-based model for safety semantics. *Int. J. Comput. Math.*, 88(4):665–694, 2011.

[11] A. Moser, C. Krügel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 231–245. IEEE Computer Society, 2007.

[12] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: algorithms and applications. In J. Field and M. Hicks, editors, *POPL*, pages 137–150. ACM, 2012.

[13] M. Ward. The Closure Operators of a Lattice. *Annals of Mathematics*, 43(2):191–196, 1942.