Action Languages and the Mitigation of Malware

Marcello Balduccini and Spiros Mancoridis

Computer Science Department
Drexel University
Philadelphia, PA 19104 USA
{mbalduccini,mancors}@drexel.edu

Abstract Automating malware mitigation requires taking into account potentially intricate dependencies among the system's components, understanding potential side-effects of the possible actions, and ensuring that required system functionalities are preserved. Answers still need to be found for fundamental questions: What does it mean to mitigate malware? When can one claim that malware has been mitigated? What are the side-effects of a mitigation strategy?
This paper aims to demonstrate that techniques from reasoning about actions and change can provide the means to create a precise characterization of the notion of mitigation and by defining corresponding algorithms. The key observation underlying our work is that a computer system can be viewed as a dynamic system, i.e., a system whose state changes over time. Taking this perspective makes it possible to leverage the techniques for reasoning about actions and change to model, and reason about, system components and malware. Furthermore, efficient computation can be achieved by relying on encodings based on Answer Set Programming.

1 Introduction

Malicious software (malware) is software designed to infiltrate, damage, or otherwise compromise computer systems and networks. Experience has demonstrated that there is no perfect prevention mechanism from malware. Any system worth compromising will eventually get infected. Thus, automated tools have been developed to detect malware's landing on a system. Of those, antiviruses are easily eluded by modern polymorphic malware[1] and are ineffective against attacks exploiting zero-day vulnerabilities[2]. Behavioral detection tools, which monitor the system against a learned behavioral model, have better success. Once malware is detected and classified, the next logical step is mitigation, i.e., the task of eliminating or isolating malware on an infected system. A mitigation tool would receive information about family and features of detected malware from the detection and classification tools, and compute suitable mitigation strategies.

While automated detection and classification have been extensively studied [14,6,5,4], mitigation has, however, received far less attention. Yet, mitigation is possibly the most complex of these tasks, requiring taking into account intricate dependen-

[1] Polymorphic malware is such that its bytecode can change without affecting the algorithm. This limits the effectiveness of detection techniques based on signatures of the binary.

[2] Zero-day vulnerabilities are only known to the potential attackers. They are dangerous because programmers cannot work on fixing them until after the attacks have already begun.

cies among system components, understanding the potential side-effects of the mitigation actions, and ensuring that system's functionality is preserved.

This paper aims to advance research on mitigation by proposing a mathematically-precise characterization of the notion and corresponding algorithms. The key observation that underlies our work is that a computer system can be viewed as a dynamic system, i.e., a system whose state changes over time. Taking this perspective enables leveraging the results of decades of research on action languages and on reasoning about actions and change [9,3] to model, and to reason about, the dependencies among system components and malware. Efficient automated computation is achieved by relying on Answer Set Programming (ASP) techniques [8]. In line with our previous work [2], this paper demonstrates that ASP-related techniques can be instrumental to the study of reasoning as it occurs in everyday life and to its precise characterization.

Let us start by providing an intuition of how mitigation works and of its challenges. Consider a simplification of the *Kimsuky Operation*[3], in which a system has been infected with two malware instances: a command-and-control malware (CC) and a key-logger (KL). CC has the purpose of enabling the remote execution of commands by an external attacker. It periodically connects to a server to check if the attacker has submitted any request for command execution. If a request exists, CC executes the corresponding commands on the local computer. KL, on the other hand, has the purpose of stealing potentially valuable information. It collects key-presses and other user interface interactions, and sends them to a remote server, where they will be analyzed to extract private data such as authentication credentials. Let us assume that a malware detection and classification component has already identified the presence of CC and KL on a system and provided some information about them. For instance, the component may have determined that CC connects to port 2222 of a remote host in domain "atkr.to," and deactivates itself if the remote host cannot be reached. The exact address of CC's remote host, however, is unknown. KL continuously captures user interface interactions and forwards them to a host at a known address (we will call it *kl-addr* for simplicity). The remote port used for the connection is unknown. If the remote computer is unreachable, KL stores the information locally and uploads it as soon as the opportunity presents itself. How are these two malware instances to be mitigated?

Mitigation is often reduced to installing software updates that patch vulnerable services. This simplistic solution will obviously fail in the all-too-frequent cases in which updates are not yet available or the updates break compatibility with other system components. Additionally, the installation of updates may require a computer reboot, which might be unacceptable if users are actively using the computer or if the computer performs critical functions that cannot be interrupted.

A more sophisticated approach relies on leveraging the available knowledge about the malware and changing the state of the system to render the malware ineffective or to remove it altogether. For example, one way to mitigate KL may involve modifying the configuration of the computer's firewall to block traffic toward address *kl-addr*. Although the malware will remain running on the computer, its key functionality – the exfiltration of data – will be rendered inoperative. The side-effects of mitigation

[3] An actual cyber attack described in http://securelist.com/analysis/57915/the-kimsuky-operation-a-north-korean-apt and part of a cyber espionage campaign against South Korea.

strategies must be carefully considered, though. In the case of KL, since the port on the remote server to which the malware connects is not known, the firewall will have to be configured to block all traffic toward *kl-addr*. This may be acceptable if the remote computer is known to have only a malicious purpose. Nowadays, however, computers used in attacks are often legitimate systems that have been infected with malware. Many of the functions they perform are still perfectly legitimate. Thus, blocking all traffic toward it may be an unacceptable side-effect of the mitigation strategy. Imagine what would happen if the computer we are trying to defend were part of the network of a service provider, and the mitigation strategy caused it to block all traffic toward paying customers.

Thus, a successful mitigation strategy needs to act on other aspects of the system. For instance, it may be possible to kill the process(es) owned by the malware in order to disable it. Suppose the detection and classification system determined that KL belongs to a malware family whose main process respawns automatically when killed. Clearly, killing the malware's processes will not help. However, suppose that KL's operations rely on the existence of a certain file, *kl-file*, on the file system of the local computer. For example, *kl-file* could be a library needed by KL. Once KL is running, *kl-file* is locked by KL's main process and cannot be deleted. However, one could disable KL by killing its main process and immediately deleting *kl-file*, before KL has time to respawn and lock the file again.

As we hope this example has demonstrated, mitigation is indeed a challenging task. So far, little research has been conducted on automating it. In [7], the authors propose an ontology of information security knowledge. The ontology is at a rather high level of abstraction and does not address the automation of the mitigation process. In [10], fairly detailed models are described by means of impact dependency graphs and applied to the problem of situation assessment, but not to mitigation. [13] also focuses on situation assessment and covers tools and applications that can be (manually) used to mitigate malware. Finally, [11] presents interesting preliminary results on applying knowledge representation to cyber security, but does not address the problem of mitigation. In conclusion, answers still need to be found for fundamental questions: What does it mean to mitigate malware? When can one claim that malware has been mitigated? What are the side-effects of a mitigation strategy?

In this paper, we (1) present a framework that enables answering these questions by reducing the task of mitigation to the study of the properties of a declarative model of computer system and malware; (2) show that the computations can be automated by translation to ASP; (3) provide an empirical evaluation of the approach on a collection of non-trivial scenarios. Because our work is the first attempt of its kind, it is focused on abstract models of systems and malware. Based on previous experience with ASP-based reasoning, transition to actual systems and use within actual defense systems is achievable and will be addressed in the coming phases of the project. It is worth stressing that our approach yields models of computer systems and malware that are independent of the specific reasoning task considered, and can thus be easily reused and expanded.

The paper is organized as follows. Related background is provided next. Section 3 discusses the definition of state-based mitigation. The next sections describe our formalization of computer systems and malware behavior and our approach to automating mitigation by reducing it to ASP-based reasoning. Section 6 reports on the experimental evaluation of the implemented system. Finally, we draw conclusions and discuss future work.

2 Answer Set Programming and Dynamic Domains

Let us begin by defining the syntax and semantics of ASP [8,12]. Let Σ be a signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A (basic) literal is an atom a or its strong negation $\neg a$. A *rule* is a statement of the form: $h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$ where h (the head) and l_i's (the body) are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, must believe h. Symbol $\leftarrow$ is omitted if the body is empty. Rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$ are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that $\{l_1, \ldots, \text{not } l_n\}$ must not be satisfied. A rule containing variables is interpreted as the shorthand for the set of rules obtained by replacing the variables with all the possible ground terms. A *program* is a set of rules over Σ. A consistent set S of literals is closed under a rule if $h \in S$ whenever $\{l_1, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$. Set S is an answer set of a *not*-free program Π if S is the minimal set closed under its rules. The reduct, Π^S , of an arbitrary program Π w.r.t. S is obtained from Π by removing every rule containing an expression $\text{not } l$ s.t. $l \in S$ and by removing every other occurrence of $\text{not } l$. Set S is an answer set of Π if it is the answer set of Π^S. For a convenient representation of choices, in this paper we also use *constraint literals* [12], which are expressions of the form $m\{l_1, l_2, \ldots, l_k\}n$, where m, n are arithmetic expressions and l_i's are basic literals. A constraint literal is satisfied w.r.t. S whenever $m \leq |\{l_1, \ldots, l_k\} \cap S| \leq n$. Constraint literals are especially useful to reason about available choices. For example, a rule $1\{p, q, r\}1$ intuitively states that exactly one of $\{p, q, r\}$ should occur in every answer set. Recent advances in solving technology have also allowed for an efficient support of aggregates, i.e., arithmetic functions over sets of literals, and of optimization problems. An aggregate that we use later, for example, is of the form $\#sum\{1, X \ : \ p(X)\}$, and intuitively corresponds to the count of all X's such that $p(X)$ holds. Similarly, later we use a statement $\#minimize\{X@1, p \ : \ p(X)\}$ to specify that any answer set found should minimize the count of X's such that $p(X)$ holds. We refer the reader to `https://www.mat.unical.it/aspcomp2013/ASPStandardization` for more information on these constructs.

For the formalization of computer systems, we use techniques from reasoning about actions and change. *Fluents* are first-order terms denoting the properties of interest of the domain (whose truth value typically depends upon time). For example, $locked\_by(f_1, p_2)$ may represent the fact that f_1 is locked by process p_2. A fluent literal is either a fluent f or its negation $\neg f$. *Elementary actions* are also first-order terms. For example, $lock(p_2, f_3)$ may mean that f_3 is being locked by process p_2. A *compound action* is a set of elementary actions, denoting their concurrent execution. A set S of fluent literals is *consistent* if $\forall f, \{f, \neg f\} \not\subseteq S$ and *complete* if

$\forall f, \{f, \neg f\} \cap S \neq \emptyset$. Fluents are further distinguished in *inertial*, whose truth value persists over time, and *positive* (resp., *negative*) *defined fluents*, whose truth value defaults to false (resp., true) in every state. The set of the possible evolutions of a domain is represented by a *transition diagram*, i.e., a directed graph whose nodes – each labeled by a complete and consistent set of fluent literals – represent the states of the domain, and whose arcs – labeled by sets of actions – describe *state transitions*. A state transition is identified by a triple, $\langle \sigma_0, a, \sigma_1 \rangle$, where σ_i's are states and a is a compound action (see Figure 1). Transition diagrams can be compactly represented using an indirect encoding based on the research on action languages [9]. In this paper, we adopt the variant of writing such encoding in ASP – see, e.g., [1]. The encoding relies on the notion of a *trajectory* $\langle \sigma_0, a_0, \sigma_1, a_1, \ldots \rangle$, i.e., a path in the transition diagram. The states in a trajectory are identified by integers (0 is the initial state). The fact that a fluent f holds at a step i is represented by atom $h(f, i)$, where relation h stands for *holds*. If $\neg f$ is true, we write $\neg h(f, i)$. Occurrences of elementary actions are represented by an expression $o(a, i)$ (o stands for *occurs*). ASP rules (also called *laws* in this context) describe the effects of actions. An *action description* AD is a collection of such rules, together with rules formalizing the default behavior of inertial and defined fluents. The transition diagram, $\mathcal{T}(AD)$, corresponding to an action description AD is characterized unambiguously by the answer sets of $AD \cup gen$. Set gen contains the rules $1\{h(F,0), \neg h(F,0) : inertial(F)\}1$ and $\{o(A,S) : action(A)\} \leftarrow step(S)$, where relations $action$ and $step$ define ranges for actions and steps. Intuitively, the first rule "generates" all possible initial states and the second rule "generates" all possible occurrences of actions.

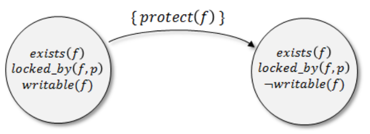

Figure 1. A possible state transition

3 State-Based Mitigation

In this section, we provide a precise characterization of possible definitions of mitigation. The aim is to enable answering the fundamental questions: What does it mean to mitigate malware? When can one claim that malware has been mitigated? What are the side-effects of a mitigation strategy? To our knowledge, this is the first such characterization, and is made possible by our choice to view mitigation as reasoning over dynamic domains.

Below, we make the simplifying assumption that, while information about the features of malware may be incomplete (e.g., we may not know which remote port KL connects to), the current state of the system and the effects of the available actions are fully known.

Let A_c be a set of rules describing the behavior of the components of a computer system, e.g., the effect of the firewall's configuration on network traffic or the effect of the actions for manipulating the file system. For the scenario from Section 1, A_c might contain a rule:

$$h(locked\_by(F,P), S+1) \leftarrow h(active(P), S), o(lock(P,F), S).$$

The rule intuitively describes the effect of a process locking a file. Let set A_m describe the behavior of malware and its relation to the computer system. For example, the fact that KL from Section 1 locks file *kl-file* when it respawns may be represented in A_m by:

$$h(locked\_by(\textit{kl-file}, P), S + 1) \leftarrow h(exists(\textit{kl-file}), S), o(respawn(M), S).$$

Finally, let $\mathit{\Gamma}$ denote a sequence of *observed states* $\langle \gamma_{-n}, \ldots, \gamma_{-2}, \gamma_{-1}, \gamma_0 \rangle$, intuitively denoting past observed states of the system, of which γ_{-n} is the oldest and γ_0 is the current state. Given $\mathit{\Gamma}$, $\mathit{\Gamma}(i)$ denotes element γ_i. Note that$\mathit{\Gamma}$ is not required to be a complete history, that is, the system may have evolved through other states between $\mathit{\Gamma}(i)$ and $\mathit{\Gamma}(i + 1)$. A *system description* is then defined as the tuple $\langle A_c, A_m, \mathit{\Gamma} \rangle$. At the core of our approach is the following definition of the notion of mitigation strategy:

Definition 1. *Given a system description $\langle A_c, A_m, \mathit{\Gamma} \rangle$, a* mitigation strategy *is a trajectory $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ in the transition diagram $\mathcal{T}(A_c \cup A_m)$ whose initial state is $\mathit{\Gamma}(0)$ and whose end state, σ_n, is a* safe state.

This definition provides a precise answer to the question "What does it mean to mitigate malware?" However, how should the notion of a safe state be defined? Next, we provide an answer to this second question, which also helps answer the fundamental question: When can one claim that malware has been mitigated? As part of the description of the behavior of malware, we assume that A_m defines the (known) conditions under which a malware becomes inactive, i.e., fluent $active(m)$ becomes false. In practice, these conditions will be provided by the classification tool. Thus, we state that:

Definition 2. *A state σ is a* strict safe state *if, for every malware m, $\neg active(m) \in \sigma$.*[4]

Although appealingly simple, this definition is in some cases too restrictive. Recall that, in the scenario from Section 1, CC could be rendered ineffective (but not inactive!) by disabling outbound traffic toward domain "atkr.to" and port 2222. Suppose that is the only way to mitigate the malware. Certainly, it is not reasonable to conclude that no safe state can be reached. Thus, we introduce in A_m a fluent $effective(m)$, intuitively meaning that m is effective (i.e., can perform its essential functions), and assume that A_m includes rules stating the conditions under which malware is rendered ineffective. We can now define:

Definition 3. *A* relaxed safe state *σ is a state in which, for every malware m, $\{\neg active(m), \neg effective(m)\} \cap \sigma \neq \emptyset$.*

A further elaboration of the notion of safe state is obtained by observing that the two definitions given above neglect to consider the side-effects of the mitigation strategy: for example, a strategy that blocks all traffic to a certain host, as discussed in Section 1, may well lead to a strict or relaxed safe state, but may be unacceptable in practice if the computer system is expected to provide (legitimate) services to remote users. So, here we turn our attention to the question "What are the side-effects of a mitigation strategy?" To model this notion, we begin by explicitly modeling the services that the computer system is required to provide. Fluents $active$ and $effective$ are extended in a natural way. Thus, we have:

[4] For simplicity, this definition does not consider malware that may infect the computer during the execution of the mitigation strategy.

Definition 4. *A* practical safe state *is a relaxed safe state σ such that, for every service s, $\{active(s), effective(s)\} \subseteq \sigma$.*

Mitigation strategies that lead to end states satisfying the above definitions are called, respectively, *strict, relaxed, and practical mitigation strategies*. In practice, it is useful to find mitigation strategies that result in the best possible state, be it a strict safe state or even a state that purposely includes disrupted computer services, if that is the only option. In the following definition, let $\prec$ denote some given ordering of states, so that $\sigma \prec \sigma'$ if, intuitively, σ is more desirable than σ'.

Definition 5. *Let S be a set of states. A state $\sigma \in S$ is* maximally safe in S w.r.t. $\prec$ *if $\forall \sigma' \in S, \sigma' \not\prec \sigma$.*

Definition 6. *Given a system description $\langle A_c, A_m, \Gamma \rangle$ and an ordering $\prec$, a* maximally safe mitigation strategy w.r.t. $\prec$ *is a trajectory $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ in $\mathcal{T}(A_c \cup A_m)$ whose initial state is $\Gamma(0)$ and whose end state, σ_n, is maximally safe among the end states of the trajectories with initial state $\Gamma(0)$.*

The actual specification of the ordering depends on one's purpose, but what is important is that this definition allows one to impose arbitrary constraints on the side-effects of the mitigation strategies. For example, one can define $\prec$ in order to limit the disruptions on the legitimate services offered by the system (see Section 5 for an example). In other cases, one may want to consider file content, as a strict mitigation strategy that formats the computer and reinstalls everything from scratch – albeit completely successful in removing the malware – is likely unacceptable from a practical perspective.

One potential issue with maximally safe mitigation is the lack of constraints on what the end state will be like. In practice, it is desirable for the end state to be one users are familiar with. Thus, our final refinement attempts to "roll back" the system to a familiar state.

Definition 7. *Given a system description $\langle A_c, A_m, \Gamma \rangle$ and an ordering $\prec$, a* roll-back mitigation strategy *is a a trajectory $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ in $\mathcal{T}(A_c \cup A_m)$ whose initial state is $\Gamma(0)$ and whose end state, σ_n, is maximally safe in Γ w.r.t. $\prec$.*

It is important to note that, while in traditional roll-back mechanisms there exists a predefined sequence of actions (typically, removal of software) that takes the system back to a previous state, our definition of roll-back mitigation strategy enables the search over arbitrary sequences of actions and is thus more flexible in responding to the changes operated by malware. Restricting the search to observed states has the practical advantage of reducing the mitigation strategies to be evaluated, while ensuring that the resulting state is likely familiar to a user, given that the computer system was in that state earlier.

4 Formalization of Computer Systems and Malware

The *Theory of Computer Systems and Malware* is a collection of modules, called theories, describing the behavior of the various components. The theories and their connections are illustrated in Figure 2. The main features of each theory are discussed next. At the current stage, we have focused on a rather high-level formalization of the components, but still sufficient for solving challenging mitigation problems such as the one described in Section 1. The *Theory of Firewalls* formalizes the behavior of a sim-

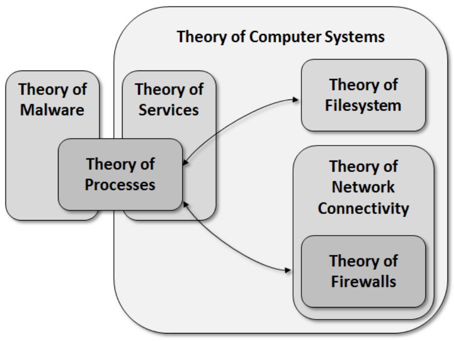

Figure 2. The Theory of Computer Systems and Malware

ple computer firewall capable of blocking various kinds of outbound traffic (IP/port pair, all ports of an IP address, all traffic to a port, all traffic to an internet domain) and inbound traffic for a given port. The firewall is formalized as a dynamic domain, whose state is characterized by inertial fluent $blocked(D, O)$, which states that traffic is blocked in direction D (i.e., in or out) to/from network object O (where O may be, for example, an IP address or an IP/port pair). The actions of the domain are of the form $block(D, O)$, where O and D are as above. The direct effects of the actions are formalized by dynamic laws. For example, the *dynamic law* [9] stating that the effect of action $block(Dom, out)$ is to block traffic toward domain Dom is (here and below, domain predicates are omitted whenever appropriate):

$$h(blocked(out, Dom), S + 1) \leftarrow \ domain(Dom),\ o(block(out, Dom), S).$$

State constraints [9] are used to propagate the effect of traffic blocks to any related network object. For example, the following state constraint says that traffic toward a host is blocked if traffic toward a domain the host belongs to is blocked:

$$h(blocked(out, H), S) \leftarrow host(H), in\_domain(H, Dom), h(blocked(out, Dom), S).$$

Relation in_domain, assumed to be part of the problem specification, formalizes membership of hosts to domains.

The *Theory of Network Connectivity* builds upon the Theory of Firewalls to capture network connectivity as the combined effect of the computer's firewall and of external devices. The state of network connectivity is characterized by negative defined fluent $reachable(D, O)$, where O and D are as above, and by fluents defining the physical state of the network, such as $plugged(lan\_cable)$ and $on(gateway)$, which state, respectively, that the network (LAN) cable is plugged into the computer and that the gateway connecting the computer to the internet is turned on. Dynamic laws define the effect of actions that change the physical state of the network, such as $unplug(lan\_cable)$ and $turn\_off(gateway)$. State constraints determine under which conditions network objects are unreachable. The following rules state that an internet object is unreachable if related traffic is blocked at the firewall level, and that all network objects are unreachable if the cable is not plugged in:

$\neg h(reachable(D,O),S) \leftarrow\ h(blocked(D,O),S).$
$\neg h(reachable(D,O),S) \leftarrow direction(D), network\_object(O), \neg h(plugged(lan\_cable),S).$

It is worth noting the flexibility afforded by the use of a negative defined fluent, allowing one to focus on the description of cases under which reachability is disrupted and relying on the fact that all other network objects will be assumed to be reachable by default.

The Theory of Filesystem describes how the properties of the files in a filesystem change over time in response to typical filesystem-related actions. Files are identified by atoms of the form $file(F)$, intuitively stating that F is a file. For every file F, the theory defines inertial fluents $exists(F)$, $readable(F)$ and $writable(F)$, with the obvious meanings. At this stage, we do not consider read/write access rights that depend on the identity of the user (e.g., allowing a file to be readable by its owner and nobody else), but it would not be difficult to extend the theory accordingly. Additionally, we use fluent $locked\_by(F,P)$ to denote the fact that file F is locked by a process P (processes are discussed later). The state of the domain is affected by two actions: $protect(F,A)$, which protects file F from reading ($A = read$) or writing ($A = write$), and $delete(F)$, which causes F to be deleted. The latter action is formalized by the rules:

$\neg h(exists(F),S+1) \leftarrow o(delete(F),S).$ $\quad\leftarrow o(delete(F),S),\ \neg h(exists(F),S).$
$\leftarrow o(delete(F),S),\ h(locked\_by(F,P),S).$ $\quad\leftarrow o(delete(F),S),\ is\_a(F,system\_file).$

The last two *executability conditions* [9] ensure that files that are locked by a process and system files cannot be deleted.

The *Theory of Processes* describes general characteristics common to all processes (i.e., running programs) of a computer system. The characterization includes inertial fluent $active(P)$, which states that process P is active (i.e., running), and negative defined fluent $effective(P)$, which states that the process is able to perform its key functions (see Section 3). The determination of when processes become active/inactive and effective/ineffective is for the most part process-dependent, and is delegated to the process-specific theories described later. The main exception is the formalization of the direct and indirect effects of killing a process, captured by the rules:

$$\begin{array}{l}\neg h(locked\_by(F,P),S) \leftarrow\ file(F),\ o(kill(P),S),\ \neg h(active(P),S).\\ \neg h(active(P),S+1) \leftarrow\ o(kill(P),S).\\ \leftarrow o(kill(P),S), \neg h(active(P),S).\end{array}$$

The next component of the formalization is the *Theory of Services*, which provides a representation of the services offered by a computer system. The theory is geared towards the features needed for reasoning about malware mitigation, and thus abstracts from details such as the computation performed, and memory or processor usage. Rather, a service is modeled as a process that receives requests on a specified port and becomes ineffective if network traffic to that port is blocked. The corresponding rule is:

$$\neg h(effective(Serv),S) \leftarrow\ on\_port(Serv,Port),\ \neg h(reachable(in,Port),S).$$

Atom $on\_port(Serv,Port)$ indicates the port on which requests are received by a service. It is worth noting how the Theory of Services builds upon the network connectivity state provided by the Theory of Network Connectivity. The theory also defines a positive defined fluent $disrupted(Serv)$, which determines when a service is disrupted:

$$\begin{array}{l}h(disrupted(Serv),S) \leftarrow\ \neg h(active(Serv),S).\\ h(disrupted(Serv),S) \leftarrow\ \neg h(effective(Serv),S).\end{array}$$

This fluent is used during the computation of mitigation strategies to determine possible disruptions to the services as side-effects of the strategies.

Finally, the *Theory of Malware* formalizes the available knowledge about the behavior of malware. As we discussed earlier, this is the first attempt at automating mitigation that we are aware of; thus, we restrict the scope of our work slightly and focus specifically on CC and KL malware. The choice of CC and KL is due to their diffusion and to the potentially significant consequences. Although the theory of malware would have to be extended in order to deal with other kinds of malware (e.g., malware that does not use network communications), the framework itself is general enough to accommodate other kinds of malware.

In our approach, a malware instance is formalized as a process with certain specific properties. It is conceivable that, in practice, this theory will eventually consist of a collection of modules corresponding to the various malware families and provided by malware detection and classification tools present on the computer. The theory includes two positive defined fluents: $mitigated(M)$, which states that malware M is currently mitigated, and $respawning(M)$, stating that the malware is in the process of respawning (see Section 3). A malware is declared mitigated if it is ineffective (fluent literal $\neg effective(M)$), or inactive ($\neg active(M)$) and not currently respawning ($\neg respawning(M)$). Changes to fluents $active(M)$ and $effective(M)$ depend on the specific malware family and malware instance. For example, in the fictitious (but not unrealistic) scenario described earlier, a command-and-control malware becomes inactive if it is unable to reach the host that controls it. This is captured by the state constraint:

$$\neg h(active(CC), S) \leftarrow is\_a(CC, cc), commanded\_by(CC, O), \neg h(reachable(out, O), S).$$

Atom $is\_a(CC, cc)$ specifies that CC is a command-and-control malware. Relation is_a, defined as part of the problem description, constitutes a simple ontology of malware. Atom $commanded\_by(CC, O)$, also provided by the problem description, indicates the network object that controls the malware. Using a network object enables the theory to deal with incompleteness of information about the commanding host. For example, if all that is known is that the malware contacts some unidentified host on port 2222, this information can be modeled by a network object corresponding to port 2222. The definition of fluent $reachable(D, O)$ in the Theory of Network Connectivity ensures that the effect of traffic blocks and of physical network conditions is properly reflected even in the absence of complete information.

Additionally, the theory prescribes that any kind of malware becomes inactive if any of the files essential to its functioning are removed:

$$\neg h(active(M), S) \leftarrow is\_a(M, malware),\ essential\_file(M, F, exist),\ \neg h(exists(F), S).$$

Atom $essential\_file(M, F, exist)$, defined in the problem description, indicates that it is essential for file F to exist. Similar atoms indicate that a file must be readable or writable.

To model respawning malware, the theory defines two special triggered actions (actions that are only executed when triggered by some other action): $init(M)$, indicating that the malware is initializing, and $respawn(M)$, which states that the malware is completing the respawning process. The first action is triggered by a $kill$ action, as defined by the trigger:

$$o(init(M), S+1) \leftarrow\ respawns(M),\ can\_respawn(M, S),\ o(kill(M), S).$$

Atom $respawns(M)$, assumed to be defined in the problem description, determines whether the malware is a respawning one. Auxiliary relation $can\_respawn(M, S)$ is used to check whether the malware can actually respawn – for example, malware cannot respawn if files essential to its functioning are missing. Action $respawn(M)$ is triggered by the execution of $init(M)$. This sequence of two triggered actions is used to model the fact that, typically, processes take a non-negligible amount of time to respawn. Once $respawn(M)$ is executed, the malware's process is considered fully running, with all the corresponding ramifications. For example, all essential files are locked, if they exist:

$$\begin{aligned} h(locked\_by(F, M), S + 1) \leftarrow\ & essential\_file(M, F, \_),\ h(exists(F), S), \\ & can\_respawn(M, S),\ o(respawn(M), S). \end{aligned}$$

The usual domain-independent rules formalize the inertial axiom, the default behavior of defined fluents and the transitive closure of relation is_a. This concludes the presentation of the Theory of Computer Systems and Malware. Although the formalization is relatively simple, it is remarkably effective at capturing the elements essential for reasoning about rather challenging mitigation problems.

5 Mitigation Module

The reasoning algorithm responsible for finding mitigation strategies given a specific problem description relies on the generate-and-test approach typical of ASP reasoning modules. The following choice rule states that a mitigation strategy consists of a sequence of at least one action per time step, until the mitigation goal has been achieved: $1\{o(A, S) : action(A)\} \leftarrow step(S), \text{not } goal(S)$. The mitigation strategies discussed in Section 3 can be implemented in ASP in a rather straightforward manner. For example, consider the mitigation module, M_r, consisting of the previous rule together with:

$$\begin{array}{l} \neg goal(S) \leftarrow\ is\_a(M, malware),\ \neg h(mitigated(M), S). \\ \neg goal(S) \leftarrow\ is\_a(M, service),\ h(disrupted(M), S). \\ goal(S) \leftarrow\ step(S),\ \text{not } \neg goal(S). \qquad goal \leftarrow goal(S). \qquad \leftarrow \text{not } goal. \end{array}$$

Upon inspection of M_r, it can be seen that our approach enables reducing mitigation to the well-studied task of ASP planning. Hence, it is not difficult to prove the following result:

Theorem 1. *let $\mathcal{S} = \langle A_c, A_m, \Gamma \rangle$. The answer sets of $A_c \cup A_m \cup \Gamma(0) \cup M_r$ are in 1-to-1 correspondence with the relaxed mitigation strategies for $\mathcal{S}$.*
Proof. (Sketch) *The thesis can be proven by observing the similarity between M_r and ASP planning modules and then applying the proving techniques used for planning.* $\square$

For the story from Section 1, M_r is capable of producing the mitigation $o(kill(cc_1), 0)$, $o(block(\textit{kl-addr}, out), 1)$: CC is killed, while KL is made ineffective by blocking all traffic to the remote host. This mitigation has the drawback of leaving KL active. To improve on this, we refine M_r to find maximally safe mitigation strategies. Let $\prec_a$ be defined so that $\sigma \prec_a \sigma'$ iff: $|\{m \,|\, \neg mitigated(m) \in \sigma'\}| > 0$, or $|\{s \,|\, disrupted(s) \in \sigma'\}| > 0$, or $|\{m \,|\, active(m) \in \sigma\}| < |\{m \,|\, active(m) \in \sigma'\}|$. Intuitively, $\prec_a$ aims at mitigating all malware, while no service is disrupted, and as many malware instances as possible are made inactive (as opposed to only ineffective).

Consider the mitigation module, M_m, containing M_r and the rules ($ls(S)$ denotes the upper bound of the range of S):

$$active(T) \leftarrow T = \#sum\{1, M : h(active(M), LS), is\_a(M, malware), ls(LS)\}.$$
$$\#minimize\{A@1, active : active(A)\}.$$

Proposition 1. *Let $\mathcal{S} = \langle A_c, A_m, \Gamma \rangle$. The answer sets of $A_c \cup A_m \cup \Gamma(0) \cup M_m$ are in 1-to-1 correspondence with the maximally safe mitigation strategies for $\mathcal{S}$ w.r.t. $\prec_a$.*

M_m finds a more desirable strategy[5] than the previous one: $o(kill(kl_1), 0)$, $o(delete(\textit{kl-file}), 1)$, $o(block(\text{“}atkr.to\text{”}, out), 2)$. The strategy blocks traffic to the domain of CC's remote host, forcing CC to deactivate itself. To deactivate KL, the strategy takes advantage of KL's dependence on *kl-file*. File *kl-file* is locked by KL, though. So, KL is killed, removing the lock, and then the file is immediately removed, preventing KL's respawning.

6 Experimental Evaluation

The goal of the experimental evaluation was to determine performance sensitivity to problem size. The experiments were conducted on a computer with an Intel i7 processor at 2.6 GHz and Fedora Core 19. The solver was restricted to one core, 8 GB RAM and no virtual memory. The solver used was CLINGO-4.4.0 with no custom options.

We randomly generated $1,000$ problems representing infected systems. Problem parameters were the number of malware instances (1 to 5) and that of legitimate services (1 to 39 in increments of 2). The ranges were selected to be representative of a medium-sized system. 10 instances were generated for each pair of problem parameters. Every malware instance was, with equal probability, a command-and-control or a keylogger. To ensure some interdependency among malware instances and system, each problem used a pool of no more than 5 essential files, remote hosts, and ports. Files could be regular or system files with equal probability.

For the experimental setup, the Theory of Computer Systems was as described earlier. The Mitigation Module was designed to find, for every problem, a maximally safe mitigation strategy w.r.t. $\prec_a$, with the additional constraint that, in case of ties, the strategy involving the smallest number of actions was selected. Mitigation strategies were allowed to be no more than 3 steps long, with an arbitrary number of concurrent actions per time step. We evaluated the execution time of the solver (grounding and solving combined), whether or not a solution was found, the *relaxation index* (i.e., how many malware instances were made ineffective but not inactive), and the number of actions.

A summary of the analysis of the experimental results is shown in Figure 3. Detailed results for the experiments with 4 and 5 malware instances are provided in Figure 4. The analysis shows that the average execution time for a given pair of problem parameters was always under or near 1 second. The average across the $1,000$ problems was 0.2 seconds. 863 instances were successfully solved: a success rate close to 90%. In 585 instances out of the 863 solved, all malware was rendered inactive; in 221, the relaxation

[5] In fact, M_m returns multiple solutions, because nothing prevents the generation of irrelevant actions. See next section for an improvement on this.

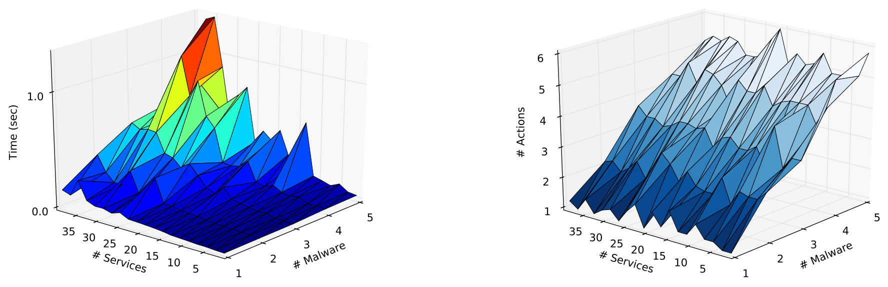

Figure 3. Execution time *(top)*; number of actions *(bottom)*

index was 1; in 55, it was 2; and only in 2 cases 3 malware instances were left active but ineffective (these cases were for problems with 5 malware instances). Figure 3 also reports the average number of actions for a given pair of parameters; the average number across the spectrum was 3.2 and the absolute maximum 9.

7 Conclusions

This paper aimed at taking a first concrete step toward the precise characterization and automation of malware mitigation, and answering fundamental questions about mitigation. The approach is based on a formalization that views computer system and malware as a dynamic domain, represented using techniques from reasoning about actions and change, action languages, and ASP. This yields reasoning-task-agnostic models that naturally capture the ramifications of the mitigation actions. We also demonstrated that the mitigation process can be automated by means of ASP-based reasoning, and stated the correspondence between two mitigation modules and the corresponding definitions of mitigation. Remarkably, our approach makes it possible to reduce the task of mitigation to the well-studied task of planning. Finally, we reported on the empirical evaluation of our approach on a collection of non-trivial mitigation scenarios. Although experiments on actual systems are needed before strong claims can be made, the results, including execution times consistently 1 second or less, can indeed be considered encouraging.

Simplifying assumptions were made, including the completeness of the information about the system's state (with the exception of information about malware characteristics). Future work will aim at lifting them. Additionally, the Theory of Computer Systems should be made more detailed, e.g., by modeling realistic file access rights and file content. Experiments on actual systems are also needed, as well as integration of our techniques with information from actual malware classification tools.

Param. Pair	# Malware	# Services	Time (sec.)	# Actions	Relax. Index
61	4	1	0.034	5.3	0.6
62	4	3	0.043	4.3	0.6
63	4	5	0.040	3.7	0.6
64	4	7	0.083	4.3	0.2
65	4	9	0.067	4.3	0.0
66	4	11	0.081	4.3	0.7
67	4	13	0.125	4.2	0.4
68	4	15	0.094	4.0	0.5
69	4	17	0.114	4.6	0.1
70	4	19	0.283	4.3	0.1
71	4	21	0.218	4.3	0.7
72	4	23	0.204	4.6	0.9
73	4	25	0.214	3.7	0.2
74	4	27	0.506	4.3	0.1
75	4	29	0.593	4.6	0.2
76	4	31	0.891	4.7	0.3
77	4	33	0.742	4.2	0.4
78	4	35	1.076	4.8	0.4
79	4	37	0.528	4.3	0.4
80	4	39	0.610	4.3	0.3
81	5	1	0.041	5.8	0.6
82	5	3	0.052	5.0	0.6
83	5	5	0.104	5.1	0.8
84	5	7	0.070	4.9	0.8
85	5	9	0.095	4.9	0.5
86	5	11	0.117	5.5	0.9
87	5	13	0.576	5.1	1.0
88	5	15	0.200	5.0	0.6
89	5	17	0.252	5.3	0.3
90	5	19	0.454	4.9	1.1
91	5	21	0.344	6.0	0.6
92	5	23	0.412	5.6	1.0
93	5	25	0.295	5.0	0.2
94	5	27	0.772	4.9	0.4
95	5	29	0.688	4.9	1.1
96	5	31	0.410	5.3	0.8
97	5	33	0.905	5.4	0.4
98	5	35	1.333	5.2	1.3
99	5	37	1.300	5.3	0.6
100	5	39	1.128	5.1	0.9

Figure 4. Experimental results for 4 and 5 malware instances (averages over 10 instances)

References

1. Balduccini, M., Gelfond, M., Nogueira, M.: A-Prolog as a tool for declarative programming. In: Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000). pp. 63–72 (2000)
2. Balduccini, M., Girotto, S.: Formalization of Psychological Knowledge in Answer Set Programming and its Application. Journal of Theory and Practice of Logic Programming (TPLP) 10(4–6), 725–740 (Jul 2010)
3. Baral, C., Son, T.C.: Formalizing sensing actions – a transition function based approach. Artificial Intelligence Journal 125(1–2), 19–91 (Jan 2001)
4. Canzanese, R., Kam, M., Mancoridis, S.: Toward an automatic, online behavioral malware classification system. In: Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) (2013)
5. Cesare, S., Xiang, Y., Zhou, W.: Malwise: An effective and efficient classification system for packed and polymorphic malware. IEEE Transactions on Computers 62(6), 1193–1206 (2013)
6. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 5–14. ESEC-FSE '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1287624.1287628
7. Fenz, S., Ekelhart, A.: Formalizing Information Security Knowledge. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS'09). pp. 183–194 (2009)
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
9. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on AI 3(16), 193–210 (1998)
10. Jakobson, G.: Mission Cyber Security Situation Assessment Using Impact Dependency Graphs. In: Proceedings of the 14th International Conference on Information Fusion (FUSION). pp. 1–8 (2011)
11. Kandefer, M., Shapiro, S.C., Stotz, A., Sudit, M.: Symbolic Reasoning in the Cyber Security Domain. In: Proceedings of MSS 2007 National Symposium on Sensor and Data Fusion (2007)
12. Niemelä, I., Simons, P.: Logic-Based Artificial Intelligence, chap. Extending the Smodels System with Cardinality and Weight Constraints, pp. 491–521. Kluwer Academic Publishers (2000)
13. Ruspini, E.H., Corkill, D.D., Powell, G.M., Das, S., Kokar, M.M., Salerno, J., Kadar, I., Blasch, E.: Issues and Challenges of Knowledge Representation and Reasoning Methods in Situation Assessment (Level 2 Fusion). In: Proc. SPIE 6235, Signal Processing, Sensor Fusion, and Target Recognition XV (2006)
14. Ye, N., Li, X., Chen, Q., Emran, S.M., Xu, M.: Probabilistic techniques for intrusion detection based on computer audit data. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 31(4), 266–274 (2001)