Semi-automated wrapping of defenses (SAWD) for cyber command and control

Marco Carvalho*, Thomas C. Eskridge*, Michael Atighetchi†, Captain Nicholas Paltzer‡
*Harris Institute for Assured Information
Florida Institute of Technology
Melbourne, FL U.S.A
Email: mcarvalho,teskridge@fit.edu
†Raytheon BBN Technologies
Cambridge, MA U.S.A
Email: matighet@bbn.com
‡U.S. Air Force Research Laboratory
Rome, NY U.S.A.
Email: nicholas.paltzer@us.af.mil

***Abstract*—In this paper we introduce SAWD, a semi-automated approach for wrapping dynamic and moving target cyber defenses for cyber command and control operations. SAWD provides an interactive user interface that enables the intuitive description (either explicitly or by example) of the requirements, pre-conditions, and steps for defense installation, configuration, and operation in multiple operational environments. The defense wrapping produces a semantic representation of the defense, as well as the code necessary to manage the defense, which enables its seamless on demand deployment and control across multiple operating systems. SAWD is a collaborative effort between Florida Institute of Technology, Raytheon BBN Technologies, and Air Force Research Laboratory.**

***Index Terms*—Computer Security, Moving Target Defense, Cyber Command and Control**

I. Introduction

In recent years, there has been a progressive trend towards dynamic and proactive network defense strategies and systems resilience. The concept of moving target defense (MTD) has recently shown increasing traction and acceptance as a mechanism designed to increase the asymmetry between attacker and defenders [1], [2]. MTDs are relatively complex defensive strategies that are aimed at creating a perceived change in the target, effectively increasing the uncertainty of the attack surface exposed to an adversary. If properly controlled and coordinated, these kinds of defenses arguably provide a powerful capability for proactive defense infrastructures, and could become a game changer in applied cyber defense [3], [4]. There are still, however, multiple challenges to be overcome as we envision the deployment and use of MTDs.

Some of these challenges are related to the management and control overhead of these kinds of defenses, which often require manual installation, configuration and control, with support for runtime (and contextual) strategy adaptation and control. Another set of challenges deals with the complexities associated with understanding the cost and effectiveness of defense deployments and configurations [5].

We introduce a tool for the semi-automated wrapping of defenses (SAWD) and other cyber packages. SAWD has three roles:

1) Internally, it semantically describes a software package and its full lifecycle on a host— including procedures for installing, starting, stopping, and uninstalling— using the Web Ontology Language (OWL). It also describes the control methods available on the software at runtime, and their parameters. See Sections III-A, III-B, and III-D.
2) To an end user, it provides a graphical interface where the representation described above can be created for a software package. We refer to this process as *wrapping*, and to the end result as *wrappers*. See Section IV.
3) On a target host, it interprets a software wrapper and "resolves" the abstract procedure vocabulary into concrete, operating system–specific commands. It also starts an agent within a command-and-control infrastructure that serves as a runtime shim to the underlying package. See Sections III-E and III-F.

The objective of this work is to address the deployment and control challenges associated with cyber sensors, actuators, and defenses in order to facilitate the integration of these tools across a network infrastructure. In this paper, we present related work and its shortcomings, our approach to software wrapping, a walkthrough with an example wrapper, and possibilities for future work.

II. Related Work

Many tools already exist in the software deployment space and are well-known by the system administration community. Tools like Ansible[1], Chef[2], and Puppet[3] provide "IT

[1] https://www.ansible.com/
[2] https://www.chef.io/
[3] https://puppetlabs.com/

automation" and allow administrators to configure servers with configuration scripts, e.g. "use `aptitude` to install Java" and "template a file with these contents." These tools offer an improvement over ad-hoc shell scripts, but lack a high-level representation. Furthermore, they have no support for controlling the software they install—their role ends once the system is configured. On the other hand, our work focuses on the full lifespan of a software package, including deployment, control, and removal.

III. Approach

The goal of SAWD is to create an executable package that can install, uninstall, configure, start, stop, and control a sensor, actuator, or defense. Further, the package includes a semantic representation of the deployment steps and control methods that allow it to be fully managed by existing command-and-control infrastructures.

A. Semantic Representation

All of the information about a software package, except the code representing its control agent, is contained in an ontology. The use of ontologies to describe the software's deployment procedures enables the descriptions to be understood by any consuming tool who shares the ontology. In this paper, we only discuss the SAWD interpreter, but the description could be used, for example, to produce documentation on the software, or analyze deployment requirements (e.g. input parameters, network usage, expected permissions).

B. Abstract Vocabulary

The deployment procedures stored in the ontology do not represent concrete commands to execute, rather they outline abstract intentions. In other words, the procedures are represented as the "what", not the "how". For example, one install step may be *GetSourceHTTP, url=http://defenses.org/foo-defense.zip*. Note the difference between this and a `wget` command—since we are describing the intention as opposed to the action, we can realize the *GetSourceHTTP* abstraction across operating systems without the need to adjust our install procedure.

The current SAWD vocabulary includes 23 terms in four categories: build from source (e.g. *BuildMaven*, *GetSourceHTTP*), files (e.g. *CreateDirectory*, *FindAndReplace*), processes (e.g. *LaunchProcess*, *RestartService*), and system (e.g. *ConfigureNetworkInterface*, *CreateUser*). Each of these steps has zero or more parameters associated with it, which are similarly abstract. For example, *CreateUser* accepts two string parameters *Username* and *Password*, and one boolean parameter *IsAdminstrator*.

The vocabulary can be expanded by modifying the base ontology file in concept mapping tools such as IHMC CmapTools[4]. We consider more user-friendly approaches in Section V.

[4] `http://cmap.ihmc.us/`

C. Parameterization

With such abstract descriptions of deployment procedures, it's important to allow for rich parameterization so that the wrapper can be customized for a variety of end systems. SAWD supports strongly-typed input parameters with defaults and constraints. These parameters can be used throughout the deployment procedures. These parameters are formally described to allow an external tool, such as one performing automated experimentation, to reason about the possible values

In addition to input parameters, SAWD supports simple user-defined[5] variables. These values are not typed; they are simply replaced as-is in procedure parameters. Although user-defined variables can be overridden in the same manner as input parameters, they are typically just for the convenience of the author and do not need to be considered by a deployer.

D. Deployment Procedures

Every wrapper contains four deployment procedures, each of which is constructed using the same set of vocabulary terms described in III-B. However, each procedure has a specific intention:

Install Procedure: Describes the steps to install the software and perform any common configuration. The install procedure does not perform configuration of the software that may be specific to an installation. For example, an install procedure could download an IP hopping defense and configure an NTP server; but it should not configure a range of IP addresses to hop on. The install procedure also should not start the software. Given its responsibilities, the install procedure is usually the longest procedure in the wrapper.

Start Procedure: Describes the steps to perform specific configuration changes to the software, and then start it. The length of this procedure is proportional to the complexity of the software configuration, but it may just include modifying a configuration file and launching a process.

Stop Procedure: Describes the steps to stop the software, perhaps as simple as killing a process. Generally, the stop procedure is not considered responsible for "unconfiguring" the software—that is, it is not meant to fully reverse the start procedure. However, it may be wise for an operator to use the stop procedure to scrub sensitive information from the system.

Uninstall Procedure: Describes the steps to reverse the install procedure. Unlike the stop procedure, the uninstall procedure should perform a thorough cleanup of the changes made during installation. Semi-automation of the uninstallation is discussed as future work in Section V.

For convenience, the wrapper may include additional files, such as source code or configuration templates, which will be deployed to a temporary directory at installation. The user can reference these files during any of the deployment procedures.

E. Runtime Control

There is a fifth major component of the wrapper that is not a procedure. The user is able to define methods, and their

[5] Here, "user" refers to the author of the wrapper.

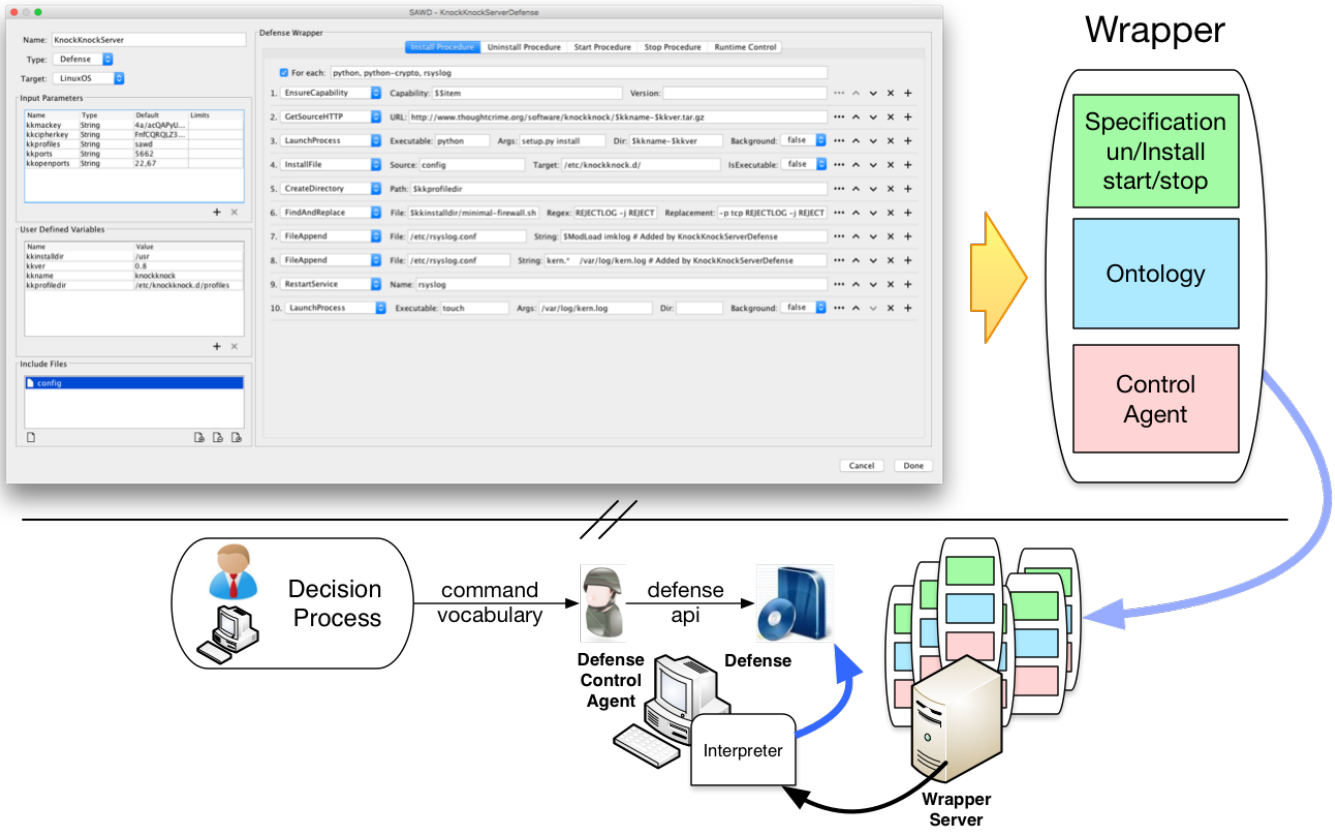

Fig. 1. Overview of the SAWD Approach

parameters (including types and constraints), that represent controllable capabilities of the software. For example, software that uses a shared key may have a *ChangeKey(String Key)* method.

When the user defines a control method, two changes are made to the wrapper:

1) The wrapper ontology is updated with the semantic descriptions of the method. Like the other descriptions in the ontology, these entries allow the command-and-control infrastructure—or other tools—to understand the capabilities of the software without concerning itself with specific APIs and protocols.
2) Skeleton agent code is created for the method, which the user can fill out in the GUI (see Figure 3). Alternatives to requiring user-written code are discussed as future work (Section V).

Libraries may be attached for the control agent to import; they will be included in the wrapper. The agent is compiled when the wrapper is packaged and launched when the operator starts a wrapper, after the start procedure completes. The agent is killed when the wrapper is stopped, before the stop procedure begins.

In the current prototype, SAWD generates appropriate baseline code so that the agent plugs into Mission-aware Infrastructure for Resilient Agents (MIRA) [4], which supports command-and-control capabilities. In addition to being directly controllable from other agents and services within the MIRA domain, external HTTP APIs are exposed that allow non-MIRA entities to interact with the control agent.

F. Interpreter

SAWD relies on an interpreter to resolve the procedures described in Section III-D. The interpreter considers the abstract vocabulary, the current operating system, input parameters, and user-defined variables; and then makes a decision about how to execute each step in the procedure. For example, the *EnsureCapability* action will rely on `apt-get` on Debian-based distributions of Linux, `yum` on Red Hat–based distributions, and `choco`[6] on Windows. This enables the same wrapper to be deployed to multiple operating systems and with various configurations. If necessary, an operator may fine tune the concrete implementation used on a target host.

The library of implementations can be expanded by writing small subclasses of abstract classes that map to the abstract vocabulary, e.g. `AptGetEnsureCapability` extends `AbstractEnsureCapability`, that know which parameters to expect from the *EnsureCapability* term.

[6]Chocolatey (`choco`) is a command line package manager for Windows. See `https://chocolatey.org/`.

G. Distribution

Although the distribution of wrappers isn't the focus of our work, our prototype lifecycle involves an FTP server to which authors upload wrappers and from which operators download wrappers. In our implementation, the download and execution of the wrapper is performed by an existing command-and-control infrastructure. However, this process could be executed by a human user, a scheduled job, or some other means.

H. Profiling

Once a wrapper has been installed and started, it can be deployed in an experimentation framework and used by an experimentation controller [6] to systematically characterize the defense. The experimentation framework supports characterization of security and cost aspects associated with wrapper deployment and use. To profile security, the experiment controller automatically builds a complete model of the system, including the wrapper installation, containing hosts, processes, flows, and users. In addition, experiments can measure time-to-success for attacks. To profile overhead of the wrapper on application performance and throughput, the experimentation framework provides an emulation component for generating traffic patterns associated with workflows across multiple distributed components. To quantify the cost in terms of system overhead, the experimentation framework measures CPU and memory footprints. Comparisons across various configurations of the wrapper enable systematic characterization of security and cost impacts.

IV. Example Wrapper

In this section, we describe the process of wrapping a single packet authorization (SPA) defense called KnockKnock. SPA is a more advanced sibling of port knocking (PK). In both cases, a service lives behind a deny-all firewall, which is opened temporarily when a special signal is received from a client. PK clients typically send a sequence of packets, e.g. TCP SYN on ports 7000, 8000, and 9000, which is highly subject to replay. On the other hand, SPA uses cryptographic payloads to transmit authorizations.

We only discuss wrapping the server, although we have also produced a wrapper for clients to install the knocking utilities, and a SOCKS proxy that knocks on behalf of client processes.

A. Preliminary Steps

The first step to wrapping a defense is outlining the responsibilities of each deployment procedure, and the control methods. For KnockKnock, this includes:

Install: Install dependencies such as Python and Rsyslog. Run the KnockKnock setup script. Enable kernel logging for Rsyslog (used to detect knock packets).

Start: Configure profiles for specific hosts. Launch the daemon that looks for knock packets and manages the firewall.

Stop: Remove profiles. Clear the firewall.

Uninstall: Remove KnockKnock code and configurations. Disable kernel logging for Rsyslog.

Based on this outline and knowledge of the KnockKnock configuration file, we can describe the input parameters to the wrapper such as knocking port, keys to use, and ports to keep open, e.g. 22 for SSH and 67 for DHCP.

B. Deployment Procedures

The full install procedure contains 10 steps, shown in Figure 2, several of which are reproduced and explained below:

- *GetSourceHTTP* with *URL* set to zip file containing the KnockKnock source. This file will be downloaded and extracted to a temporary directory.
- *LaunchProcess* with *Executable*="python", *Args*="setup.py install", and *Directory* set to the path created in the previous step.
- *InstallFile* with *Source*="config" and *Target*="$kkdir". "config" is a default configuration file we have attached to the wrapper. The source of *InstallFile* does not have to be included in the wrapper; it can exist anywhere on the target host and may be retrieved in another step. "$kkdir" references a user-defined variable used for convenience throughout the wrapper procedures, which we have set to "/etc/knockknock.d".
- *FileAppend* with *File*="/etc/rsyslog.conf" and *String*= "kern.* /var/log/kern.log". This enables Rsyslog to log kernel events, which KnockKnock follows to detect knocks.

Observe how these steps, while technical in nature, do not include implementation details. KnockKnock does require `iptables`, but this wrapper allows installation on any suitable operating system.

For brevity, we don't include the start, stop, or uninstall procedures in this paper.

C. Runtime Control

Although KnockKnock has no APIs for runtime control, we define a method called *setDelay* which takes one integer parameter, *delay*, with a default of 15, a minimum of 1, and a maximum of 300. This method will update the configuration file for KnockKnock and restart the daemon process.

Once we define this method, SAWD generates the code stub for a Java `setDelay()` method with the appropriate parameter, `Integer delay`. We can populate this method in the wrapper editor as shown in Figure 3.

D. Execution

The completed wrapper is an executable JAR file, which takes as arguments one of the install procedures (e.g. "install") and key-values pairs to override input parameters, if necessary. The distribution of this executable is discussed in Sectioned III-G. To install KnockKnock with a profile called "demo" and a knocking port of 4424:

```
java -jar KnockKnockServerDefense.jar install \
  kkprofiles=demo kkports=4424
```

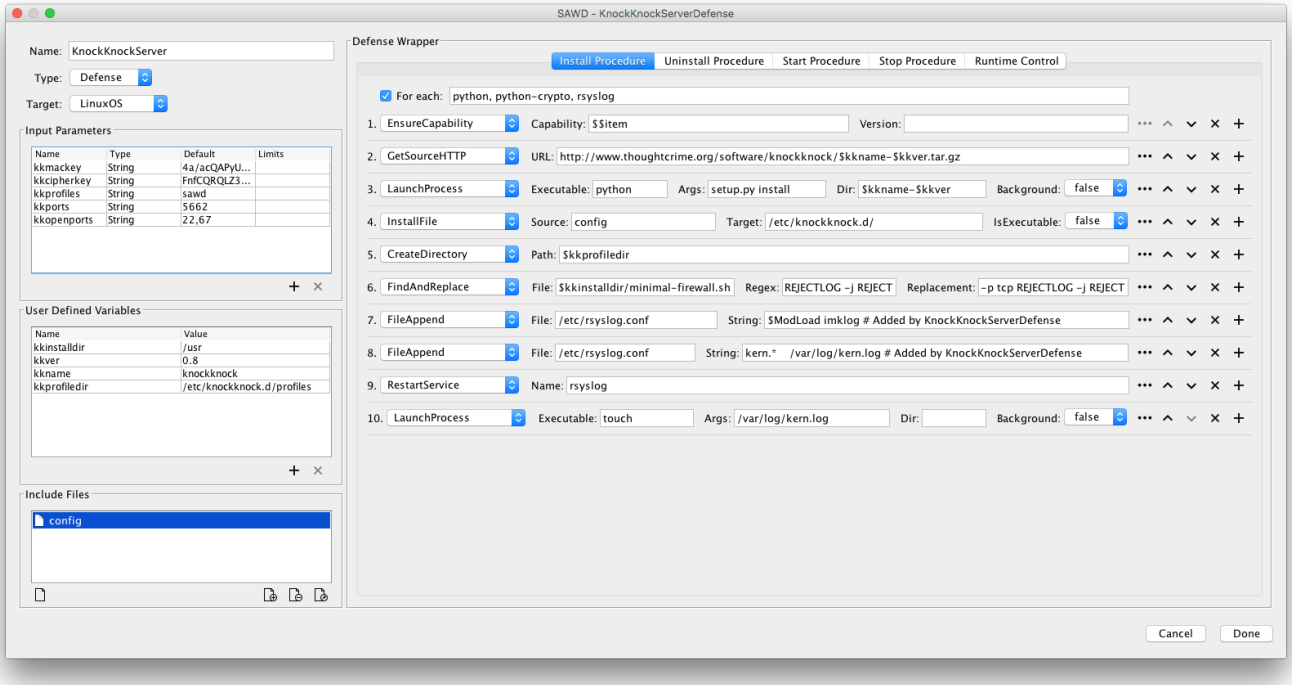

Fig. 2. Install procedure for the KnockKnock server.

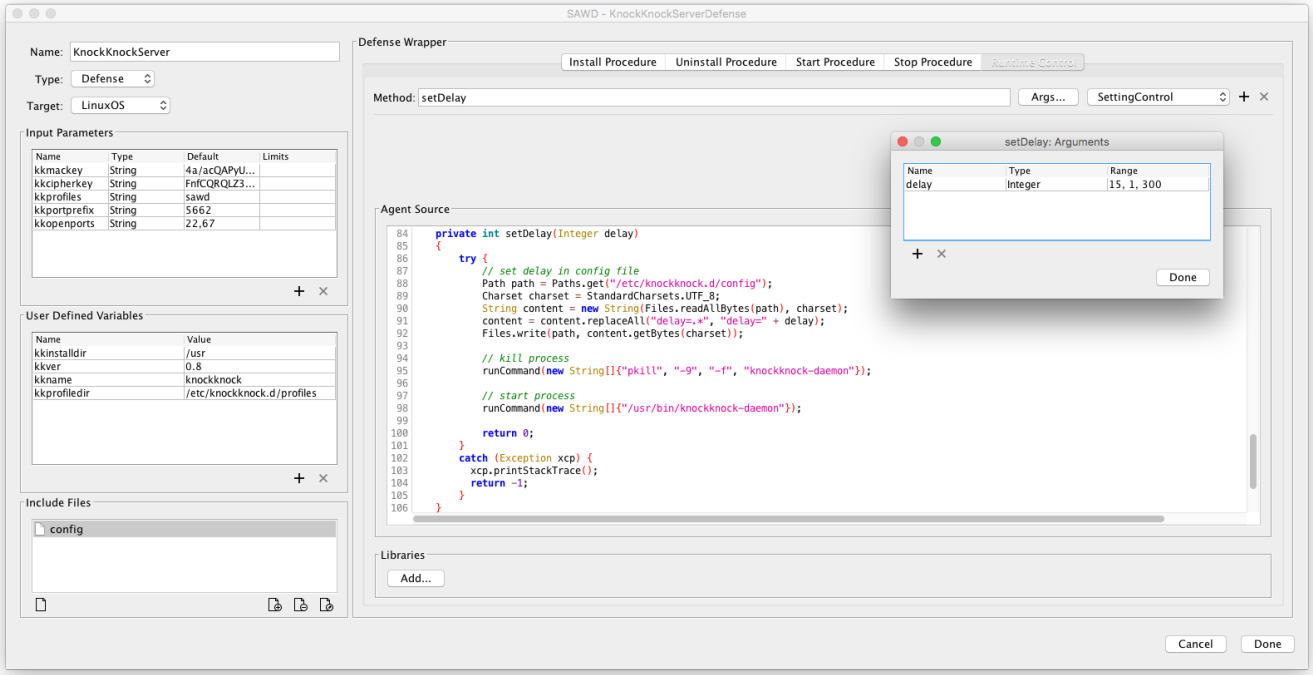

Fig. 3. Runtime control configuration for the KnockKnock server. SAWD has generated the `setDelay` method stub, which the user has populated with code to adjust the configuration and restart the server.

Running the start procedure will also add the control agent to the local MIRA environment, if one exists, at which point an operator can adjust the delay of the KnockKnock server.

V. Future Work

Through our experience wrapping various defenses and other software packages, we discovered two challenges. First, the code-your-own approach to the control agent, while allowing for the greatest flexibility, is infeasible to some users. Even for seasoned programmers, it can be a time-consuming process. We are interested in developing an abstract vocabulary for control methods, similar to the vocabulary for deployment procedures. In the case of KnockKnock, we could conceivably describe *setDelay* as two steps, *ReplaceInFile* and *RestartProcess*. This would greatly improve the usability of SAWD to less-technical users.

Second, although the graphical interface is approachable for new users, it can impede those who have become experienced with SAWD. For these users, we would like to develop an "advanced view" where the graphical controls are mostly replaced with a text editor and a domain-specific wrapping language. For instance, using HCL[7]:

```
param "kkurl" {
  type = "String"
  default = "http://.../knockknock.zip"
}

step "download source" {
  type = "GetSourceHTTP"
  url = "$kkurl"
}
```

In addition to these two interests, we would also like to explore options for editing the abstract vocabularies, specifically how to write these abstractions and their implementations in code and how to append them to the base ontology. Finally, we are considering automated wrapper cleanup, such as automatically restoring firewalls and removing unused dependencies.

VI. Conclusions

In this paper, we discussed SAWD, a tool that allows cyber sensors, actuators, and defenses to be wrapped into a self-contained, executable package for installation, runtime control, and removal. The wrappers contain ontological descriptions of the software, its deployment procedures, and its control capabilities, which allows it to be managed by command-and-control infrastructures. SAWD effectively addresses the deployment challenges discussed in Section I by providing an automated but highly-configurable interface to virtually any software package.

Acknowledgment

This research project is sponsored by the U.S. Department of Defense. Any opinions, findings and conclusions or recommendations presented in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

The authors thank Evan Stoner and Adrian Granados of Florida Institute of Technology for their contributions to the project and to this paper.

[7] https://github.com/hashicorp/hcl

References

[1] M. Carvalho, J. Bradshaw, L. Bunch, T. Eskridge, P. Feltovich, and R. Hoffman, "Moving target defense: A survey," IHMC and Florida Institute of Technology, Report, January 2011.

[2] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer Publishing Company, Incorporated, 2011, edition: 1st.

[3] M. Carvalho, T. Eskridge, L. Bunch, A. Dalton, R. Hoffman, J. Bradshaw, P. Feltovich, D. Kidwell, and T. Shanklin, "MTC2: A command and control framework for moving target defense and cyber resilience," in *6th International Symposium on Resilient Control Systems (ISRCS)*, San Francisco, CA, 2013.

[4] M. Carvalho, T. C. Eskridge, K. Ferguson-Walter, N. Paltzer, D. Myers, and D. Last, "MIRA: A support infrastructure for cyber command and control operations," in *3rd International Symposium on Resilient Cyber Systems*. Philadelphia, PA.: TeX Users Group, August 2015, pp. 84–89.

[5] M. Atighetchi, J. Griffith, I. Emmons, D. Mankins, and R. Guidorizzi, "Federated access to cyber observables for detection of targeted attacks," in *Military Communications Conference (MILCOM), 2014 IEEE*, 2015, Conference Proceedings, pp. 60–66. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6956738

[6] M. Atighetchi, B. Simidchieva, M. Carvalho, and D. Last, "Experimentation support for cyber security evaluations," in *11th Annual Cyber and Information Security Research (CISR) Conference*, 2016, Conference Proceedings.