


A Wingman for Virtual Appliances

Prashanth Nayak¹, Mike Hibler², David Johnson², and Eric Eide²

¹ NetApp, Research Triangle Park, NC, USA
prashant.u.nayak@gmail.com

² University of Utah, Salt Lake City, UT, USA
{hibler, johnsond, eeide}@cs.utah.edu

Abstract. Wingman is a run-time monitoring system that aims to detect and mitigate anomalies, including malware infections, within virtual appliances (VAs). It observes the kernel state of a VA and uses an expert system to determine when that state is anomalous. Wingman does not simply restart a compromised VA; instead, it attempts to repair the VA, thereby minimizing potential downtime and state loss. This paper describes Wingman and summarizes experiments in which it detected and mitigated three types of malware within a web-server VA. For each attack, Wingman was able to defend the VA by bringing it to an acceptable state.

1 Introduction

A *virtual appliance* [15], or VA, is a virtual machine that is deployed to run a specific application or service. For example, a company might use a VA containing a “LAMP stack”—Linux, Apache, MySQL, and PHP—to serve its web site. VAs are commonly assembled from large software components, and even if this software is high quality, bugs and security issues are inevitable. Research suggests that there are 6–16 bugs per thousand lines of code [12,13], leaving VAs vulnerable to run-time failures and attacks.

The simplest way to recover a compromised VA is to completely reinitialize it. This incurs downtime and loss of state, and in many situations, these costs may not be acceptable to users of the VA—especially since, until the attack vector is closed, the VA may need to be restarted continually. This paper explores an alternative recovery strategy: *online repair*. The goal of online repair is to automatically bring a running but compromised appliance to an *acceptable state*, one in which the VA can perform its primary task while also satisfying administrator-specified integrity properties.

Wingman is our prototype tool that performs online VA repair. It uses virtual-machine introspection (VMI) [6] to collect snapshots of a VA’s kernel, and it uses an expert system [14] to determine when a snapshot represents an anomaly. Wingman’s data-collection and expert-system components run outside the monitored VA, but to carry out repairs, Wingman invokes a kernel module within the VA. This design eases the implementation of complex repair actions. Wingman’s data-collection, anomaly-detection, and repair components are reusable across many different VAs. To target Wingman to a new VA, an administrator only needs to encode a set of facts about the VA’s application.

This paper describes Wingman’s design and implementation, and it summarizes experiments in which we used Wingman to successfully defend a web-server VA from three types of malware. Additional information about Wingman and its evaluation can be found in Nayak’s thesis [11]. The Wingman tool is open-source software [7].

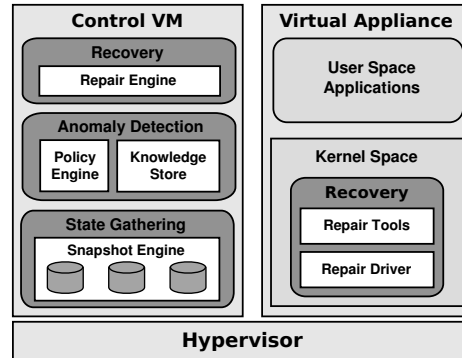


Fig. 1. Wingman’s architecture. Logical components are shown as gray rounded rectangles; software components are shown as white rectangles. Adapted from Nayak [11].

2 Design and Implementation

Figure 1 illustrates Wingman’s architecture. Its logical components are (1) state gathering, (2) anomaly detection, and (3) recovery (repair). The state-gathering and anomaly-detection components run in a *control VM* and are thus isolated from the VA being protected. The recovery component is spread across the control VM and the VA.

The discrete software components of the architecture are shown as white rectangles in Fig. 1. Wingman begins by taking a snapshot of the VA: the *snapshot engine* gathers point-in-time state snapshots of the VA’s applications, and it encodes these snapshots as “facts” in the *knowledge store*. Next, the *policy engine* uses those facts and inference rules in the knowledge store to detect anomalies. The *repair engine* uses information in the knowledge store to reason about an appropriate repair strategy and informs the *repair driver*. Finally, the repair driver invokes one or more *repair tools* to carry out the chosen repair. Wingman runs periodically, sampling the state of the VA and detecting and repairing anomalies, and thus works to keep the appliance in an acceptable state.

Wingman is built atop Xen and Linux. The snapshot engine, knowledge store, policy engine, and repair engine execute within the protected Xen dom0 (control VM). The VA’s applications, along with the repair driver and tools, execute in a Xen domU running Linux. The snapshot engine gathers state from the VA using the Stackdb [6] VMI libraries. The policy engine is an expert system built using the open-source CLIPS [14] framework. The recovery subsystem is implemented as a combination of a CLIPS-based expert system running in dom0 and Linux kernel modules in domU. The communication channel between the repair engine and the kernel-based repair driver is built with Stackdb.

2.1 State Gathering

The snapshot engine is a VMI application that collects the current state of the VA into a “snapshot.” It executes periodically in the user space of the control VM, and is thus isolated from possible malware or other problems inside the VA. The engine uses VMI libraries to extract values from the VA’s kernel data structures.

```
(task-struct
(comm "mysqld") (pid 663) (tgid 663) ; process name, ID, thread group ID
(used_superuserpriv 1) ; has superuser privileges?
(uid 108) (euid 108) (suid 108) ; real, effective, and saved user IDs
(gid 120) (egid 120) (sgid 120) ; real, effective, and saved group IDs
(fsuid 108) (fsgid 120) ; UID, GID for filesystem access
(parent_pid 1) (parent_name "init")) ; parent process ID, name
```

Listing 1. Example Base Fact for a `mysqld` Process. Adapted from Nayak [11].

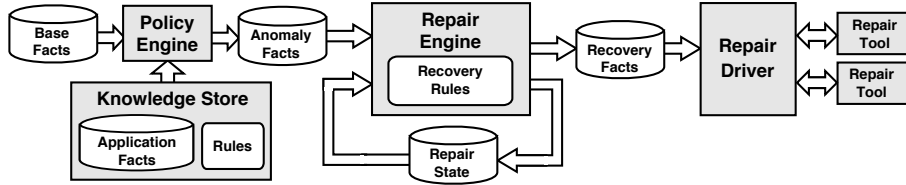


Fig. 2. Anomaly detection and recovery components. Adapted from Nayak [11].

Base facts. A snapshot contains data about each process executing in the VA: its credentials, environment variables, scheduling priority, CPU utilization, open files and network sockets, and loaded object files. A snapshot also contains system-wide information, such as CPU load, system-call entry vectors, and loaded kernel modules. All this data is encoded as a collection of records, called *base facts*. Listing 1 shows some of the key fields for an example base fact: a `task-struct` fact representing a `mysqld` process.

2.2 Anomaly Detection

The policy engine and knowledge store run in the control VM. Together, they drive the detection of problems in the VA. The policy engine is an expert system that reasons over collections of facts about the VA—facts that represent the current state of the VA as well as those describing its expected state. As illustrated on the left-hand side of Fig. 2, the policy engine takes as input (1) the *base facts* captured by the snapshot engine, (2) *application-specific facts* representing the expected state of the VA, and (3) *rules* that classify observations as expected or anomalous. From these things, the policy engine generates a set of *anomaly facts* that represent any unexpected state in the VA.

Application-specific facts. The knowledge store contains Wingman’s application-specific knowledge about the VA, provided by the VA’s creator or administrator. The application-specific facts represent the expected state of the VA in terms of kernel-visible abstractions such as processes, files, users, sockets, loaded object files (i.e., shared libraries), and kernel modules. Listing 2 shows an example set of facts that capture an administrator’s knowledge that an Apache process with the given credentials, possible parent processes, and loaded object files should be executing in the VA.

To protect a particular VA, the administrator or creator of the VA only needs to specify an appropriate set of application-specific facts. He or she can reuse facts created for similar VAs or create an entirely new set of facts. Due to its kernel-centric design, all of the other parts of Wingman are *application-independent* and are reusable for all VAs.

```

;; Declare the allowable processes in the VA.
(known-processes ... "apache2" ...)
;; Declare that Apache must be running in the VA.
(mandatory-process (name "apache2") (command /etc/init.d/apache2 start ...))
;; Declare the expected credentials of the Apache process.
(known-process-cred (name "apache2") (parent_name "init" "apache2")
  (uid 33) (euid 33) (suid 33) (fsuid 33) (gid 33) (egid 33) (sgid 33) (fsgid 33))
;; Declare the shared objects of the Apache process.
(known-objects (name "apache2")
  (object-list "apache2" "libnss_files-2.15.so" "libnss_nis-2.15.so"
    "mod_uni_mem.so" "mod_rewrite.so" "mod_alias.so" ...))

```

Listing 2. Example Application Facts for an apache2 Process

```

(defrule identify-unknown-process
  (task-struct (comm ?name) (pid ?pid) (parent_pid ?ppid))
  (known-processes $?proclist)
  (test (not (member $?name $?proclist))))
=> (assert (unknown-process (name ?name) (pid ?pid) (ppid ?ppid)))
  (printout t "ANOMALY: Unknown process " ?name " found" crlf))

```

Listing 3. Example Rule for Identifying Unknown Processes. Adapted from Nayak [11].

Rules. Inference rules use the application-specific and base facts to validate the state of the VA; they capture an administrator’s domain expertise and automate anomaly detection. Listing 3 presents a rule that identifies all unknown processes running in the VA. The clauses above “=>” are the premise, and those following it are the conclusion: if the premise is true, the conclusion is executed. The example rule matches all `task-struct` base facts against the `known-processes` application-specific fact to detect unknown processes. The technique of comparing the VA’s dynamic kernel state (base facts) to the properties of acceptable states (application-specific facts) allows Wingman to identify a large and general class of anomalies, including multiple types of malware.

Anomaly facts. Rules produce new facts about observed anomalies. In Listing 3, if any process identified by a `task-struct` is not also present in `known-processes`, the rule creates (“asserts”) an `unknown-process` anomaly fact to record the problem.

2.3 Recovery

Once an anomaly has been identified by the policy engine, Wingman’s recovery components reason about an appropriate recovery strategy and attempt to restore the VA to an acceptable state. The recovery workflow is shown on the right-hand side of Fig. 2. The repair engine is an expert system that executes in the control VM along with the snapshot engine, policy engine, and knowledge store. It takes as input the *anomaly facts* generated by the policy engine, passes them through a set of *recovery rules*, and generates a set of *recovery facts* that are used to select appropriate repair tools.

Recovery rules. Recovery rules generate recovery facts from anomaly facts, and thus suggest repairs for anomalies, but the mappings are not simply one-to-one. The recovery rules operate over additional *repair state* (facts) kept by the repair engine, such

```

(defrule kill-unknown-process
  ?f <- (unknown-process (name ?name) (pid ?pid))
  (not (exists (unkn-proc-prev-action (prev_action ps_kill | ps_kill_parent)
    (name ?name) (pid ?pid))))
=> (assert (recovery-action (func-name kill_process) (arg_list ?name ?pid)))
    (assert (unkn-proc-prev-action (name ?name) (pid ?pid) (prev_action ps_kill)))
    (retract ?f) (printout t "RECOVERY: Killing the unknown process" ?pid crlf))

(defrule kill-unknown-process_1
  ?f <- (unknown-process (name ?name) (pid ?pid))
  ?of <- (unkn-proc-prev-action (prev_action ps_kill) (name ?name))
=> (assert (recovery-action (func-name kill_parent_proc) (arglist ?pid ?name)))
    (retract ?f) (retract ?of)
    (assert (unkn-proc-prev-action (name ?name) (prev_action ps_kill_parent)))
    (printout t "RECOVERY: Killing the process its parent" crlf))

```

Listing 4. Example “Kill Unknown Process” Recovery Rules. Adapted from Nayak [11].

as the history of previous occurrences of the anomaly and recovery actions already taken. This allows the repair engine to better reason about future repair actions.

Listing 4 shows two possible recovery rules that handle unknown processes; they map `unknown-process` facts to appropriate recovery actions. The `kill-unknown-process` rule kills newly discovered unknown processes. The premise matches unknown processes for which no recovery action has already been attempted. The conclusion defines the recovery action: asserting a recovery fact that identifies the repair tool to be used (`kill_process`), asserting a repair-state fact recording that this rule has been tried, and retracting the fact that the process is unknown, since Wingman is about to kill it. It will be rediscovered in the next state-gathering iteration if the `kill_process` tool fails. The `kill-unknown-process_1` rule handles unknown processes that still exist in the VA after Wingman has tried killing them with `kill-unknown-process`. In this case, the recovery action is to kill both the process and its parent.

Repair tools. Recovery facts identify individual *repair tools*, which are then invoked via the *repair driver*. The repair driver and tools run inside the VA to simplify tool development and make complex repairs feasible. For example, it is straightforward for a tool within the VA’s kernel to start a new process or retrieve swapped-out pages, using the kernel’s own code. It is practically impossible for an agent entirely outside the VA to perform these tasks through VMI alone. Because the repair engine runs in a different VM than the repair driver and tools (Fig. 1), tool-invocation commands are sent from the repair engine to the driver through an inter-VM communication channel. The result is communicated back to the repair engine through the same channel.

Wingman’s most basic tools act on processes and their attributes. The `psaction` tool terminates a process by traversing the kernel’s process list to locate it, and then calling the Linux `force_sig` function to deliver a SIGKILL. The `ps_deescalate` tool resets a process’s credentials to specified values by modifying the process’s `cred` structure. The `kill_socket` tool shuts down the open sockets of a process, and the `close_file` tool closes its opened files. The `start_process` tool starts a user-space process.

Other tools perform more sophisticated repairs. The `system_map_reset` tool provides two functions: one to fix corrupt system-call table entries and another to fix over-

written system-call function prologues. The `trusted_load` and `start_process` tools work together to start processes in a controlled-boot environment. The `trusted_load` tool inputs the names of blacklisted objects that are not allowed to be loaded by a process created by `start_process`. This environment allows only non-blacklisted objects to load during process startup. The tool hooks the `open` and `mmap` system calls with versions that return an error when the process tries to load a blacklisted object.

The `sled_object` tool deals with malicious objects already loaded into process memory by overwriting them. It calls the Linux `get_user_pages` function to load the pages containing the code segment of an object, and overwrites all non-return instructions with `no-ops`. Thus, every function in the malicious object is “nullified”: calling them does nothing but return. This repair only works on functions with a hook-like API: i.e., that have a `void` return type and are not required to perform any action.

3 Evaluation

We deployed Wingman to monitor a web-server virtual appliance. We evaluated its effectiveness against three different types of malicious software, including a kernel rootkit, a user-space rootkit, and an application malware.

Experiment context. We ran a web-server VA on a server (64-bit 2.40 GHz quad-core Xeon E5530 CPU, 12 GB RAM) running Xen 4.1.2. Both the control VM and VA ran Ubuntu 12.04 with a Linux 3.8.0 kernel; the VA also ran the Apache 2.2 web server.

The policy in the knowledge store described the VA’s acceptable states via 103 CLIPS facts (415 lines of code). It allowed the execution of Apache and PHP processes, MySQL, NTP, and SSH daemons, and standard kernel processes (e.g., `kworker`).

Wingman’s snapshot engine pauses the VA to ensure atomic snapshots, and thus affects VA availability. The average time to capture a complete snapshot is 140.3 ms; during that time, no work is performed by the VA. Wingman does not otherwise affect the VA’s availability, since the policy and repair engines run without pausing the VA.

Kernel rootkit. A kernel rootkit is a set of malicious programs that provide continuous, unauthorized root access. The Suterusu [2] rootkit provides features including root-shell access; process, socket, and file hiding, and disabling module loading. Unlike traditional rootkits, Suterusu does not modify the function pointers in the system-call table; instead, it overwrites instructions in the prologues of the functions themselves. This allows Suterusu to evade detection by most rootkit detectors.

Suterusu hides processes by hooking the `proc_root_readdir` function of the `/proc` filesystem. (Most tools use `/proc` to list processes.) When run against the infected VA, Wingman found a Suterusu-hidden rogue process because it scans the in-kernel process list; it restored the VA to an acceptable state by killing the process. To detect file-related malicious behaviors, our policy restricted process file access. For instance, if a process violates the policy by opening `/etc/shadow`, Wingman identifies the access and closes the corresponding file descriptor. Suterusu hooks the `tcp4_seq_show` and `udp4_seq_show` functions (et al.) to hide TCP or UDP sockets. Since the VA’s policy allows only specific processes to open sockets, Wingman detects and closes Suterusu-hidden sockets. Because Suterusu hooks normal kernel functions, not system calls, Wingman cannot fully deactivate it—but Wingman *can* suppress its malicious activity.

User-space rootkit. Azazel [1] is a user-space rootkit that infects individual programs at execution time using LD_PRELOAD to ensure that its own malicious library functions override standard library functions. This library (named `libselinux.so`) provides functions to spawn root shells, hide processes, and deploy back doors.

When we applied Wingman to our infected web-server VA, it detected the malicious library and recovered over a period of repair iterations. First, Wingman identified the malicious shared object as an anomaly in `sshd`. To recover, the repair engine restarted `sshd`. In the second iteration, Wingman again detected the malicious object in `sshd`'s memory (because Azazel had overwritten the `ld.so.preload` file). This time, `sshd` was restarted in a controlled environment, but failed to run since it was not allowed to load `libselinux.so`. Finally, since previous repair efforts failed, the repair engine chose to nullify (“sled”) the object’s instructions. In subsequent iterations, if the unknown object were detected in new `sshd` processes, its instructions would be nullified immediately. The `sshd` process became unresponsive to connection requests after the object was “sledded.” Its inability to respond is an undetected anomaly—the VA was in an acceptable state by policy, but still had an anomaly. Because the back door was closed and the Apache web server was unaffected, we argue that Wingman successfully recovered the VA.

Azazel can exploit the `su` command to spawn a root shell; if this occurs, Wingman terminates the root shell. Wingman detects such privilege escalations by validating the process lineage in the snapshot against the VA’s policy. Azazel also sets up an SSH back door by preloading malicious objects during `sshd` startup. Wingman detects the compromised `sshd` process and restarts it, resulting in the `sshd` session being terminated.

Application malware. Darkleech [9] is an Apache module that injects malicious iframes into legitimate HTTP responses. The iframes redirect clients to malicious sites. Darkleech is loaded at Apache startup via the `LoadModule` configuration file command.

To evaluate Wingman’s effectiveness against Darkleech, we created a network consisting of three client hosts and our web-server VA. Each client ran a script that repeatedly made HTTP requests and checked the responses for malicious iframes. As the clients submitted requests, Wingman inspected the web-server VA every five seconds.

In the first iteration, Wingman detected an unknown shared object in the Apache processes, and attempted to restart it. Because Darkleech modified the Apache configuration to load its module at startup, the unknown module was loaded into the restarted process. Thus, in the second iteration, Wingman again detected the unknown module in Apache. Since the previous repair was ineffective, Wingman’s repair engine restarted Apache in a controlled-boot environment. As the Apache threads were killed by the repair tool, new threads were spawned by the main Apache process outside Wingman’s controlled environment. Thus, in the final iteration, Wingman found the unknown object again. This time, Wingman decided to “sled” the unknown object, replacing its instructions with no-ops. Once this occurred, the responses no longer contained malicious iframes.

Summary. Table 1 summarizes our experiments. Although these malware samples are only a handful of thousands, they are representative of common exploit techniques. Wingman detected all anomalies introduced by these samples. Although it was unable to clear the malicious object loaded by Azazel, it terminated its malicious actions. Wingman successfully mitigated all the actions of the other malware samples. We conclude that Wingman detected the anomalies in the VA and restored it to an acceptable state.

Table 1. Malware samples mitigated by Wingman. Adapted from Nayak [11].

Experiment	Mitigated	Acceptable	Outcome	
Suturusu	Process hiding	✓	✓	Process detected and terminated
	File access	✓	✓	Detected and stopped
	Network access	✓	✓	Detected and stopped
Azazel	Unauthorized objects	✓	?	Detected, mitigated, but not cleaned
	su exploitation	✓	✓	Detected and privileges restored
	SSH back door	✓	✓	Detected and terminated
Darkleech	✓	✓	Unknown object detected and nullified	

4 Related Work

Several previous research projects, such as Livewire [4], have used VMI to create intrusion-detection systems for virtual-machine guests. Livewire and Wingman both use VMI for anomaly detection, but differ in that Wingman attempts to repair anomalies. IntroVirt [8] uses VMI to detect past intrusions and prevent future exploits of known vulnerabilities. IntroVirt and Wingman provide complementary styles of “stopgap” protection to VMs: however, IntroVirt does not try to repair existing damage.

Exterior [3] allows processes within a “secure VM” to observe and manipulate the kernel state of a protected “guest VM.” Whereas Exterior enables cross-VM execution for repair, Wingman uses a combination of VMI for detection and a kernel module for guest VM repair, and places greater emphasis on *automated* detection and repair.

LKIM [10] combines measures of static data (e.g., code pages) with “contextual inspection” to check the integrity of a kernel’s dynamic data. OSck [5] enforces control-flow integrity through means such as write-protecting the kernel’s code pages and analyzing the targets of dynamic control-flow transfers. Wingman is similar to these systems in that they all inspect the dynamic data within a VM’s kernel and look for integrity violations. Unlike both LKIM and OSck, Wingman attempts to repair violations.

Nayak’s thesis [11] presents a more detailed discussion of work related to Wingman.

5 Conclusion

Our Wingman prototype tool demonstrates that automatic, online anomaly detection and repair can help to maintain an acceptable level of integrity within a virtual appliance over time. This is useful when it is important to run a VA continuously, with minimal downtime and state loss. Our evaluation showed that Wingman was able to detect and mitigate three different types of malware within a web-server VA. Although Wingman did not remove the malicious software in our experiments, it substantially and automatically reduced the malware’s harmful effects, bringing the VA back to an acceptable state without needing a human to initiate repair.

Acknowledgments. We performed our experiments on machines in the Utah Emulab testbed [16]. This work was supported in part by the Air Force Research Laboratory and DARPA under Contract No. FA8750–10–C–0242. This material is based upon work supported in part by the National Science Foundation under Grant No. 1314945.

References

1. Chokepoint: Azazel userland rootkit (Feb 2015), <https://github.com/chokepoint/azazel>
2. Coppola, M.: Suterusu rootkit (Sep 2014), <https://github.com/mnccoppola/suterusu>
3. Fu, Y., Lin, Z.: Exterior: using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. In: Proceedings VEE. pp. 97–110 (Mar 2013), doi:10.1145/2451512.2451534
4. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings NDSS. pp. 191–206 (Feb 2003), <http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf>
5. Hofmann, O.S., Dunn, A.M., Kim, S., Roy, I., Witchel, E.: Ensuring operating system kernel integrity with OSck. In: Proceedings ASPLOS. pp. 279–290 (Mar 2011), doi:10.1145/1950365.1950398
6. Johnson, D., Hibler, M., Eide, E.: Composable multi-level debugging with Stackdb. In: Proceedings VEE. pp. 213–226 (Mar 2014), doi:10.1145/2576195.2576212
7. Johnson, D., Nayak, P., Hibler, M., Burtsev, A., Eide, E.: Wingman and Stackdb software (Mar 2017), <https://gitlab.flux.utah.edu/a3/vmi>
8. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting past and present intrusions through vulnerability-specific predicates. In: Proceedings SOSP. pp. 91–104 (Oct 2005), doi:10.1145/1095810.1095820
9. Landesman, M.: Apache Darkleech compromises (Apr 2, 2013), <http://blogs.cisco.com/security/apache-darkleech-compromises>
10. Loscocco, P.A., Wilson, P.W., Pendergrass, J.A., McDonell, C.D.: Linux kernel integrity measurement using contextual inspection. In: Proceedings ACM Workshop on Scalable Trusted Computing (STC). pp. 21–29 (Nov 2007), doi:10.1145/1314354.1314362
11. Nayak, P.: Detecting and Mitigating Malware in Virtual Appliances. Master’s thesis, University of Utah (Dec 2014), <http://www.flux.utah.edu/paper/pnayak-thesis>
12. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. In: Proceedings ISSTA. pp. 55–64 (Jul 2002), doi:10.1145/566172.566181
13. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Where the bugs are. In: Proceedings ISSTA. pp. 86–96 (Jul 2004), doi:10.1145/1007512.1007524
14. Savely, R., Culbert, C., Riley, G., Dantes, B., Ly, B., Ortiz, C., Giarratano, J., Lopez, F.: CLIPS: A tool for building expert systems (May 2015), <http://clipsrules.sourceforge.net/>
15. Sun, C., He, L., Wang, Q., Willenborg, R.: Simplifying service deployment with virtual appliances. In: Proceedings IEEE International Conference on Services Computing (SCC). pp. 265–272 (Jul 2008), doi:10.1109/SCC.2008.53
16. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings OSDI. pp. 255–270 (Dec 2002), <https://www.usenix.org/legacy/event/osdi02/tech/white.html>