Chapter 2
Towards Intelligent Cyber Deception Systems

Fabio De Gaspari, Sushil Jajodia, Luigi V. Mancini, and Giulio Pagnotta

Abstract The increasingly sophisticated nature of cyberattacks reduces the effectiveness of expert human intervention due to their slow response times. Consequently, interest in automated agents that can make intelligent decisions and plan countermeasures is rapidly growing. In this chapter, we discuss intelligent cyber deception systems. Such systems can dynamically plan the deception strategy and use several actuators to effectively implement the cyber deception measures. We also present a prototype of a framework designed to simplify the development of cyber deception tools to be integrated with such intelligent agents.

2.1 Introduction

The knowledge of attackers and the sophistication of cyberattacks are constantly increasing, as well as the complexity of the cyber domain. The result of this process is that expert human intervention, even if available, is not always fast enough to deal with the speed of cyberthreats. As a consequence, cyber deception strategies aimed at hindering attackers' progress and cyber defense agents that can make autonomous decisions are receiving an increasing amount of attention [9, 17]. An important part of cyber deception is *active defense* [15, 16]. Differently from classical, reactive systems such as firewalls, IPS, and IDS, active defense tools aim to hinder attackers' progress in a proactive manner, rather than responding if and when an attack is detected. One of the most well-known examples of active

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

F. De Gaspari · L. V. Mancini · G. Pagnotta
Sapienza University of Rome, Roma, RM, Italy

S. Jajodia (✉)
George Mason University, Fairfax, VA, USA
e-mail: jajodia@gmu.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_2

defense tools are honeypots [5, 8]: mock systems designed to lure attackers in order to study their behavior and restrict their access to the real production systems. Other active defense techniques, like honeypatches [6], trick attackers into believing that their exploit was successful, but transparently redirect him to an unpatched, heavily monitored decoy system. Fake login sessions [26], mock services, and port randomization [4] aim at confounding the attacker, compromising and slowing down the reconnaissance phase. Coupling such active defense tools with autonomous, intelligent agents has the potential of greatly improving cyber defense, reducing the reliance on human intervention in response to cyberattacks.

In this chapter, we discuss intelligent cyber deception agents that can make autonomous decisions on how to counter ongoing attacks, and their integration with active defense tools. We also discuss our design of an active defense framework that allows fast prototyping of active defense tools to be integrated directly into live, production systems. The framework uses a modular approach to add and remove active defense tools, and aims to provide seamless integration with the agent to provide sensing and actuating functions.

2.2 Preliminaries

In this section, we discuss intelligent cyber defense agents and the complexity of deploying them in the context of cyber defense. We also discuss active defense techniques and the advantages it brings with respect to traditional systems.

2.2.1 Intelligent Cyber Defense Agents

An intelligent agent is an entity which takes autonomous decisions based on the observations of the current world state through sensors, and which applies actions through actuators to achieve an end goal. Agents can rely on different methodologies to produce decisions, such as knowledge-based systems [10] or machine learning techniques [24]. While research and applications of intelligent agents is already underway in multiple fields, this is not the case in cyber defense. Indeed, the realm of cyber defense introduces a number of unique obstacles that are particularly challenging, such as the extreme complexity and size of the state space (i.e., the possible states of the world the agent is monitoring). Artificial neural networks [12] and deep learning techniques [18] can potentially help to overcome these challenges. However, research in this direction is still in its early stages and deep learning is mostly used to devise new attacks [13] or for the purpose of attack detection [22, 23], rather than to plan countermeasures. Moreover, deep neural networks are subject to a new type of attack known as adversarial examples [11, 19]. Such attacks could be exploited by attackers to target the decision-making process of the agent itself, tricking it into taking decisions that are detrimental for the system. For instance, if

the agent is designed to shut down a particular service under certain severe attack conditions, an attacker could potentially craft an adversarial example that causes the agent to misclassify the current world state and erroneously shutdown the service.

2.2.2 Active Defense

Active defense is a branch of cybersecurity aimed at actively hindering attackers' progress preemptively, rather than reactively as in traditional systems [15, 16]. Indeed, active defense tools are always active, and do not rely on detection of an attack in order to function. Active defense is related to cyber deception. Most active defense tools, in fact, heavily rely on deception techniques to confound attackers and slow down their progress. The most well-known instance of an active defense tool is the honeypot [20]. Honeypots are replicas of real systems, instrumented with logging and deception capabilities such as fake services. Honeypots are designed to look like attractive targets for attackers, in order to obtain as much information from adversarial interactions as possible. However, honeypots require complex configuration and it is very hard to hide their nature to attackers, reducing their effectiveness [21]. To mitigate this drawback, recent works aim to integrate deception capabilities directly into the real production systems themselves, avoiding the issue of camouflaging altogether [9]. These systems use techniques similar to those employed by honeypots, as well as other active defense tools such as honeyfiles [7, 25] and network randomization [4], in order to heavily slow down the attacker, while at the same time increase the chances of detection.

2.3 Towards Intelligent Cyber Deception Systems

Autonomous agents require sensors and actuators to respectively measure and alter the current world state. Active defense tools are designed to interact with attackers and collect important data regarding how the attacker interacts with the system. Therefore, such tools can be extremely effective sensors for the agent. For instance, fake services can provide information regarding what type of services the attacker is looking for, as well as how he interacts with such services. Logging honeyfiles access provides detailed information regarding which types of files are interesting to the attacker, and honeypatch sensors allow the agent to isolate specific exploits used during the attack. Moreover, active defense tools can also be used as actuators: the agent can use the data generated by the sensors to dynamically reconfigure the active defense tools, in addition to dynamically deploying new tools aimed to hinder the specific pattern of the current attack.

In our preliminary work [9], we proposed an automated, cyber deception system called Attackers Hindered by Employing Active Defense, or *AHEAD*. The AHEAD

architecture describes an autonomous agent that employs an array of active defense tools as both sensors and actuators. AHEAD is comprised of an autonomous agent, the *AHEAD controller*, which manages a cluster of active defense tools, the *AHEAD Pot* as illustrated in Fig. 2.1.

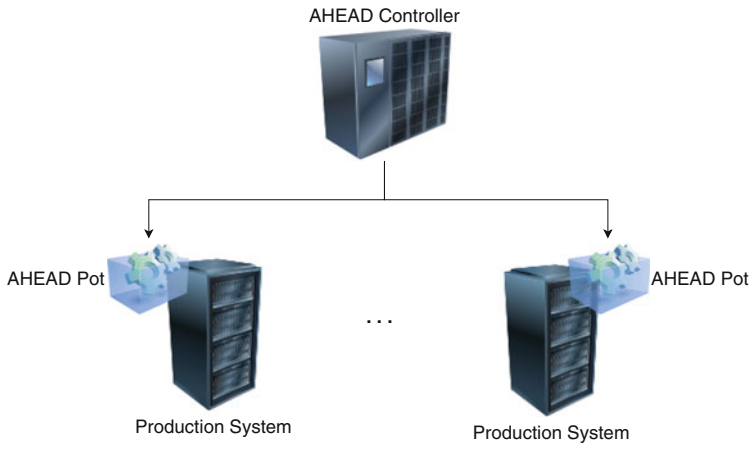

Fig. 2.1 Overview of the architecture of AHEAD

Differently from systems such as honeypots, the AHEAD Pot is deployed alongside the production services in a real system, rather than on a separate (virtual) machine. Using the AHEAD Pot to directly instrument production systems with deception capabilities allows to avoid the drawbacks of honeypots such as its ease of detection. Moreover, this design makes it harder for attackers to identify vulnerable services in the production systems, as well as providing the systems with advanced monitoring capabilities. In order to prevent the AHEAD Pot itself from becoming an attack vector, the pot is isolated from the production system through the use of container technology [2], as illustrated in Fig. 2.2. The use of containers, as well as mandatory access control techniques to limit the pot's access to the system, provides a layer of isolation and hardening against attacks directed at the pot itself. The AHEAD controller is responsible for planning the defense strategy during an attack, which is done based on the inputs from the active defense tools of the AHEAD Pot. The controller is also responsible for actuating the planned countermeasures through dynamic reconfiguration of the AHEAD Pot.

2.3.1 Usage Scenario

In this section, we describe a usage scenario of AHEAD. Let us consider an attacker who wants to attack some production systems on a target network. We distinguish the two scenarios depicted in Fig. 2.3: *(A)* a network protected by a classical

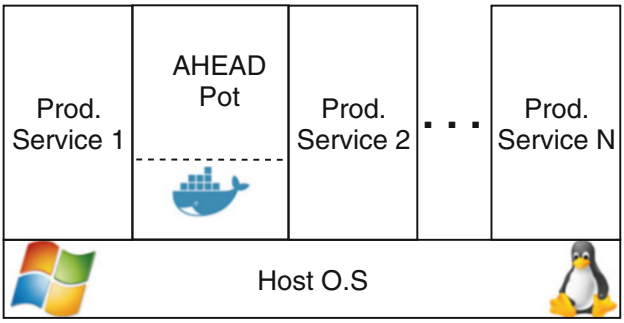

Fig. 2.2 Integration of the AHEAD Pot in the production system

honeypot and *(B)* a network protected with AHEAD. Before performing any attack, the attacker will have to perform reconnaissance on the network in order to identify valuable targets. Let us assume that, in order to reach this goal the attacker performs a network scan to identify existing systems and services.

(A) Classical Honeypot In scenario (A), the network scan will eventually reach the honeypot (step 1). At this point, if the configuration of the honeypot is realistic enough, the attacker will start attacking one of the available services provided by the honeypot. The attack is detected by the honeypot (step 2), and the Incident and Response Team (IRT) will be notified that something anomalous is going on in the network. Unfortunately, the attacker will eventually realize that the target is indeed a honeypot (step 3) and will move on to attacking one of the remaining systems (step 4). In this scenario, the limitation of the honeypot approach from the point of view of the IRT is that the interaction between the attacker and the honeypot is extremely limited in time, often in the order of seconds. Indeed, after the attacker leaves the honeypot and moves on to another system, the IRT loses the chance to monitor the attacker and devise a proper identification and defense strategy.

(B) AHEAD On the other hand, when AHEAD is employed, the attacker will have to sift through fake services and mock vulnerabilities (step 1) in order to try to compromise the production system, forcing him to interact with AHEAD for a considerably longer time (step 3.i). This provides the autonomous agent (or the IRT, if the agent is disabled) with considerably more time to act, and more information to decide how to counter the attack (step 2), as well as provide more material to analyze the strategy of the attacker after the attack has concluded (step 4). This additional information allows to improve the attribution of the attack and the security of the network and systems, adapting them to ever-evolving attack strategies. Moreover, AHEAD can also work as a deterrent. Indeed, if the attacker realizes that the real production system is heavily monitored and instrumented, he might also choose to forfeit the attack in order to protect himself (step 3.ii). In both cases, the network is protected.

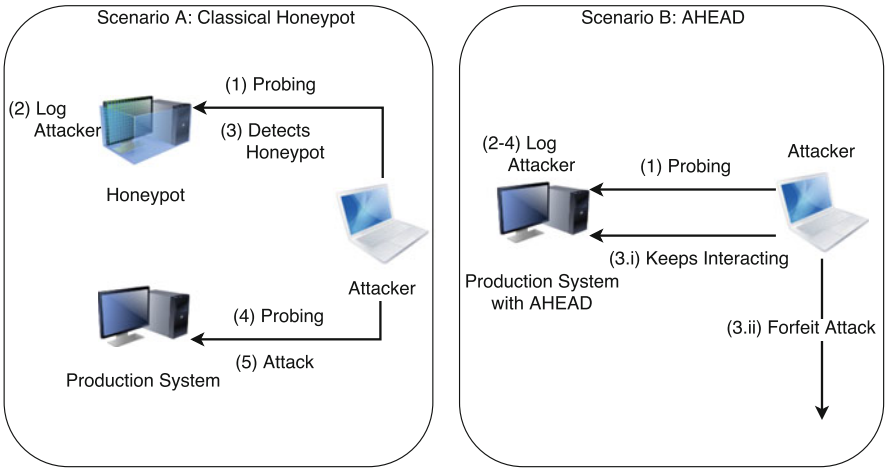

Fig. 2.3 Comparison between a classical honeypot system and AHEAD

2.3.2 The Architecture of the AHEAD System

From an architectural perspective, the AHEAD system is composed by two components: the *AHEAD Controller* and *AHEAD Pot*. The AHEAD Controller is the single point of interaction with the pots, and allows to manage the active defense tools deployed over a secure channel. The AHEAD Pot is the component effectively implementing the active defense countermeasures and is deployed on the production systems. In a real-world scenario, several AHEAD Pots are deployed in a corporate network, covering all components of the information system (see Fig. 2.4). The AHEAD Pots are encapsulated in a container and therefore do not interfere with the production services, while at the same time having a low, configurable overhead on the production system itself. Full automation of the security management of a network is a challenging task that requires gradual evolution and integration. Therefore, the AHEAD system is designed to be tightly integrated with pre-existing security information and event management (SIEM) systems, and can provide an admin interface for the security admin. The AHEAD Pots constantly send activity logs to the SIEM systems, allowing the IRT to improve other security components already deployed (e.g., intrusion detection/prevention systems, and firewall). The feedback from the AHEAD Pots is also used by the IRT to identify what additional active defense modules need to be deployed in the Pots themselves, so that the system can dynamically adapt to emerging threats. However, the final goal is for the system to be fully autonomous: the controller makes decisions on what tools should be deployed based on the current state of the world, which is reported by the tools of the pot itself as illustrated in Fig. 2.5.

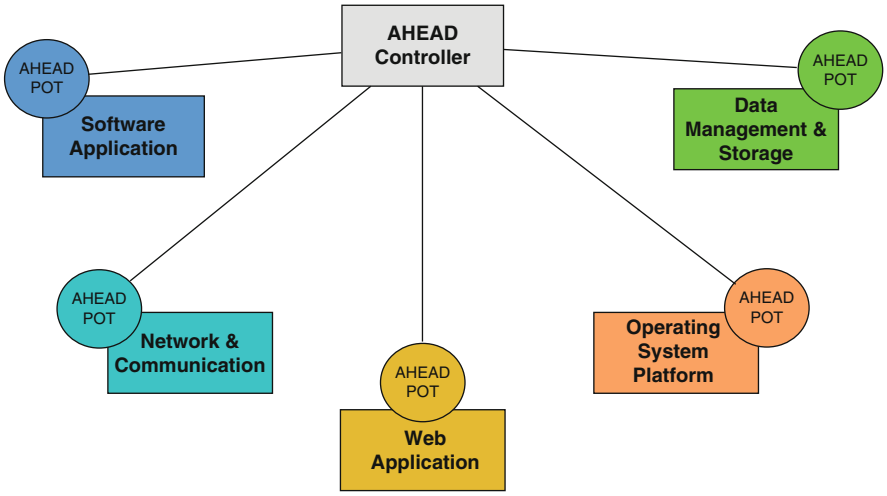

Fig. 2.4 Integration of AHEAD in the architecture of a typical corporate information system

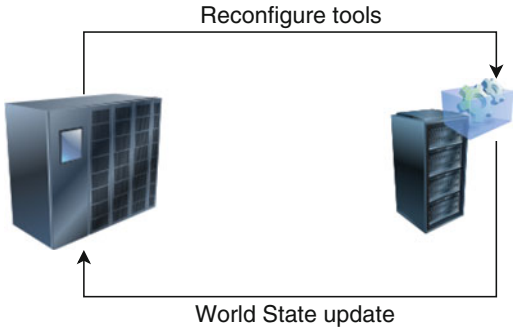

Fig. 2.5 Feedback loop of the AHEAD system. The Controller reconfigures the tools in the pot, based on the world view provided by the pot itself

2.4 Evolving the Pot: ADARCH

During the development of the AHEAD Pot prototype, we quickly realized that a unified framework for the development and integration of active defense tools was required. Existing active defense tools are implemented using a heterogeneous mix of programming languages and libraries, as well as differing architectural designs. Moreover, different tools tend to use different logging formats, sometimes custom-made, which complicates the interaction with the AHEAD Controller. Finally, having separate active defense tools makes it harder to present the attacker with a consistent view of the Pot, potentially creating side channels that allow the attacker to distinguish between services exposed by the Pot, and production services exposed by the real system. Integrating and maintaining such a diverse set of tools into a

coherent architecture would be complex and error-prone. Moreover, assessing the overall security of the system would be a daunting task, especially given the overlap in functionality between certain tools, and the code duplication ensuing from it.

In order to address the above-mentioned issues, we designed and implemented a new, cross-platform pot architecture, *ADARCH* (the Active Defense ARCHitecture), to facilitate the development of active defense tools that share a uniform architecture. The goal of ADARCH is to simplify the implementation of common functionalities of active defense tools, as well as to provide a uniform interface for the controller to interact with the tools. In particular, we identified two main functions that are used by multiple tools and that require simplification and unification: networking and logging. The first implementation of ADARCH aims to simplify and uniform the network flow management and concurrency across the tools, as well as to provide a common logging interface that enables the AHEAD Controller to more easily parse their output.

2.4.1 ADARCH Design

As we discussed in the previous section, one of the design goals of ADARCH is to simplify the development and integration of active defense tools. However, since the Pot is designed to be integrated into real production systems, it is also important to reduce the overhead introduced and the resource requirements as much as possible. To this end, we designed ADARCH and the new ADARCH Pot around a core software module written in C, and integrated a Python interpreter to facilitate the prototyping and development of active defense tools. Figure 2.6 provides a high-level overview of the ADARCH framework and the architecture of the ADARCH Pot. The core C module efficiently implements common functionalities required by multiple active defense tools, such as network connection and concurrency management, as well as provides an interface for the integrated python interpreter. Active defense tools developed with ADARCH are executed within the integrated Python interpreter, which provides them with access to the API exposed by the C core module. Moreover, ADARCH allows active defense tool developers to use a configuration file-based approach to instantiate required resources (e.g., port bindings) that are transparently handled by the ADARCH core, as well as to define triggers associated with particular functions of the tools (e.g., which function to call when a new connection is open on a port). Finally, ADARCH is designed to be cross-platform and to work with container technologies, such as Docker, to provide an additional layer of isolation to the underlying production system.

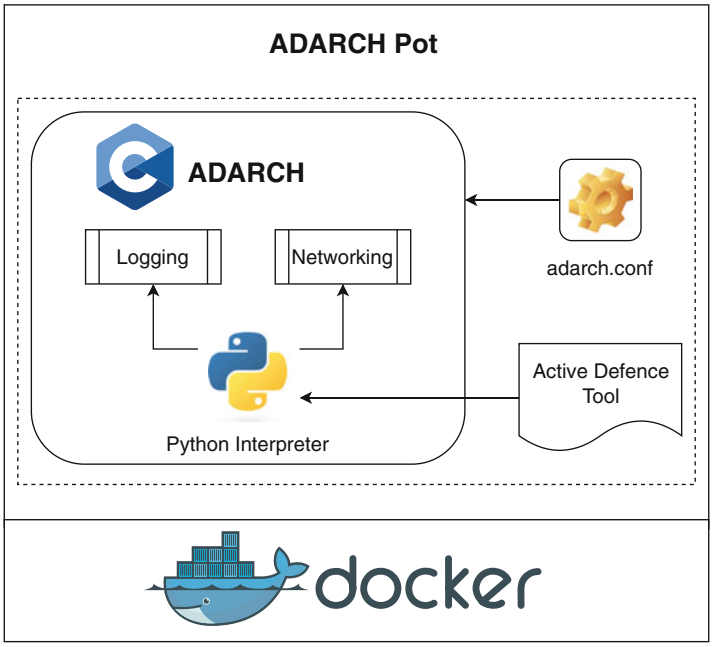

Fig. 2.6 High-level ADARCH architecture overview

2.4.2 Python Embedding and Extension

We chose Python as the programming language for the development of new active defense tools because of the large number of available libraries, the community support, and the ease of prototyping it provides. Moreover, several pre-existing active defense tools were already implemented in Python, so its choice also provides continuity for the developers. However, integrating Python with the C core of the framework to provide better efficiency and performance presented several challenges. In particular, it required overcoming the limitation of the Global Interpreter Lock (GIL) of the standard CPython interpreter, which heavily limits concurrent architectural designs [3]. The GIL prevents the Python interpreter from concurrently interpreting bytecode for different threads, effectively resulting in a sequential execution. This behavior is highly undesirable as it reduces the performance of the ADARCH Pot, and might even provide attackers with a side channel to differentiate the services exposed by the Pot from the production services [14]. To overcome these drawbacks, the interface between the Python interpreter and the C core of ADARCH was designed to manage the GIL in a fine-grained manner, releasing the lock whenever possible when an ADARCH API call is made, and trying to parallelize the execution of the various modules as much as possible.

The Python interpreter was also extended in order to expose the ADARCH's API to the tool developers. The extension of the interpreter required explicit managing

of the internal reference count of Python objects, which is used by the Python garbage collector to periodically cleanup unreferenced memory. While reference count management is generally well-documented and understood, special care was needed due to the concurrent nature of the ADARCH core. ADARCH's API, illustrated in Table 2.1, is currently minimalistic mainly due to the fact that it is updated as more active defense tools are integrated and developed into ADARCH. However, it can be easily extended to provide additional functionalities, such as filesystem management to implement integrity check tools, similarly to Artillery [1]. The API is encapsulated in a wrapper Python class, allowing developers to extend the standard API if needed.

Table 2.1 ADARCH API

API	Description
log.write	Helper function to write logs in a standard format
connection.send	Network wrapper to send data through an open connection
connection.shutdown	Network wrapper to close an open connection

2.4.3 Advantages of the ADARCH Framework

The ADARCH frameworks implements functions that are common to multiple active defense tools, such as logging and networking, and provides a transparent interface to the developer through the API and ADARCH configuration file. Allowing developers to focus on the core deception aspects of the tools, rather than having to deal with networking, threading, and synchronization, greatly simplifies and expedites the development process. Moreover, the integration of the Python interpreter allows developers to use a high-level language, further simplifying the prototyping and development of new tools, while at the same time maintaining high performance as a result of the C core module of ADARCH. The ADARCH framework is also cross-platform, working both under Linux and Windows systems, which means that active defense tools developed with ADARCH do not require additional work to be ported to different systems. Moreover, having multiple tools integrated and running within the same process space allows them to more easily share resources and interact with each other if required. Finally, ADARCH provides active defense tools with a standard format for logging, which allows for immediate integration of new active defense tools with the AHEAD Controller.

To assess the advantages of ADARCH over previous versions of the Pot, we re-implemented a popular active defense tool called Portspoof as an ADARCH module. Portspoof is an active defense tool which allows to simulate the signatures of a great number of network services. The goal of the tool is to hinder the discovery phase of an attack, forcing the attacker to perform a more thorough service scan,

and to generate much more traffic in the process. The original software, developed in C++, counts $3k$ lines of code to manage concurrent network connections using multiple threads. The corresponding ADARCH module is less than 100 lines of code written in tens of minutes and is functionally equivalent to the original tool. Moreover, our preliminary performance evaluation shows that the ADARCH tool introduces a slightly lower overhead to a production system than the original Portspoof. ADARCH extremely simplifies the development of active defense tools, heavily reducing the time required and the complexity of the code, while at the same time improving the maintainability and the security of the tools due to less code duplication. Finally, ADARCH allows to use a single, optimized instance of the Python interpreter for all active defense tools, rather than one instance per tool, further reducing system overhead when considering deployments of multiple tools.

2.5 Conclusions

In this chapter, we discussed autonomous, intelligent cyber agents and the challenges associated with their implementation. Moreover, we examine the architecture we presented in [9] and propose a new framework to develop and deploy active defense tools, ADARCH, and the new ADARCH Pot. The ADARCH framework allows to heavily simplify and expedite the development of new active defense tools and their integration in the AHEAD architecture, transparently implementing common functions required by multiple active defense tools. We discuss the advantages of ADARCH with respect to stand-alone implementations of active defense tools, and we compared the complexity of such stand-alone tools with the simplicity of an ADARCH module providing the same functionalities.

2.6 Exercises

In this section, we propose a list of exercises, in increasing order of difficulty, that can help familiarize students with the concepts presented in this chapter.

1. Identify and describe the disadvantages of the presented approach.
2. What are the trade-offs of using container technology for isolation vs. virtual machines?
3. Configure and deploy a simple honeypot on the Internet. Analyze how long attackers interact with the honeypot on average before realizing that it is not a real system.
4. What are the risks of installing active defense tools on live, production systems? How would you minimize these risks?
5. Write a simple active defense tool that can create trap files in the file system. Once opened, the files should trigger and log an alert.
6. Extend the tool described in the previous point to provide attribution capabilities.

Acknowledgements This work was partially funded by the Army Research Office under the grants W911NF-13-1-0421 and W911NF-15-1-0576, and by the Office of Naval Research under the grant N00014-15-1-2007.

References

1. Artillery. https://github.com/shoreditch-ops/artillery.
2. Docker platform. https://www.docker.com/.
3. Python Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.
4. E. Al-Shaer. *Toward Network Configuration Randomization for Moving Target Defense*, pages 153–159. 2011.
5. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 9–9, 2005.
6. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, 2014.
7. B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *Security and Privacy in Communication Networks*, pages 51–70.
8. M. L. Bringer, C. A. Chelmecki, and H. Fujinoki. A survey: Recent advances and future trends in honeypot research. In *International Journal of Computer Network and Information Security*, IJCNIS, 2012.
9. F. De Gaspari, S. Jajodia, L. V. Mancini, and A. Panico. Ahead: A new architecture for active defense. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '16, 2016.
10. J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.
11. I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, 2014.
12. M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1st edition, 1995.
13. B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz. PassGAN: A Deep Learning Approach for Password Guessing. *ArXiv*, 2017.
14. R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, 2013.
15. S. Jajodia, K. A. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and S. X. Wang, editors. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Springer, 2013.
16. S. Jajodia, K. A. Ghosh, V. Swarup, C. Wang, and S. X. Wang, editors. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 2011.
17. A. Kott, L. V. Mancini, P. Théron, M. Drašar, E. Dushku, H. Günther, M. Kont, B. LeBlanc, A. Panico, M. Pihelgas, and K. Rzadca. Initial Reference Architecture of an Intelligent Autonomous Agent for Cyber Defense. *ArXiv e-prints*, 2018.
18. Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436 EP –, May 2015.
19. N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016.
20. N. Provos. A virtual honeypot framework. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

21. N. Provos and T. Holz. *Detecting Honeypots*, chapter in book: Virtual Honeypots: From Botnet Tracking to Intrusion Detection. Addison-Wesley Professional, 2007.
22. J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.
23. S. Seufert and D. O'Brien. Machine learning for automatic defence against distributed denial of service attacks. In *2007 IEEE International Conference on Communications*, 2007.
24. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, K. Leach, Madeleineand Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. Article.
25. J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pages 116–122.
26. L. Zhao and M. Mannan. Explicit authentication response considered harmful. In *Proceedings of the 2013 New Security Paradigms Workshop*, NSPW '13, 2013.