

# ExploitFlow, cyber security exploitation routes for Game Theory and AI research in robotics

Víctor Mayoral Vilches<sup>1,2</sup>, Gelei Deng<sup>3</sup>, Yi Liu<sup>3</sup>, Martin Pinzger<sup>1</sup>, and Stefan Rass<sup>4,1</sup>

<sup>1</sup>Alpen-Adria-Universität Klagenfurt, Austria

<sup>2</sup>Alias Robotics, Spain

<sup>3</sup>Nanyang Technological University, Singapore

<sup>4</sup>Johannes Kepler University Linz, Austria

August 7, 2023

## Abstract

This paper addresses the prevalent lack of tools to facilitate and empower Game Theory and Artificial Intelligence (AI) research in cybersecurity. The primary contribution is the introduction of ExploitFlow (EF), an AI and Game Theory-driven modular library designed for cyber security exploitation. EF aims to automate attacks, combining exploits from various sources, and capturing system states post-action to reason about them and understand potential attack trees. The motivation behind EF is to bolster Game Theory and AI research in cybersecurity, with robotics as the initial focus. Results indicate that EF is effective for exploring machine learning in robot cybersecurity. An artificial agent powered by EF, using Reinforcement Learning, outperformed both brute-force and human expert approaches, laying the path for using ExploitFlow for further research. Nonetheless, we identified several limitations in EF-driven agents, including a propensity to overfit, the scarcity and production cost of datasets for generalization, and challenges in interpreting networking states across varied security settings. To leverage the strengths of ExploitFlow while addressing identified shortcomings, we present Malism, our vision for a comprehensive automated penetration testing framework with ExploitFlow at its core.

## 1 Introduction

Robots are often insecure and fully unprotected. The rationale behind this is fourfold: first, defensive security mechanisms for robots are still in their early stages, not covering the complete threat landscape. Second, the inherent complexity of robotic systems makes their protection costly, both technically and economically. Third, robot vendors are currently not taking responsibility in a timely manner, extending the zero-days exposure window (time until mitigation of a zero-day) to several years on average [1]. Fourth, contrary to the common-sense expectations and similar to Ford in the 1920s with cars, most robot manufacturers oppose or difficult robot hardware repairs and software patching. They employ planned obsolescence practices to discourage repairs and evade competition. In addition, it is observed how most manufacturers keep forwarding the cyber security problems to the end-users of these robotic machines, further obstructing the security landscape in robotics. Security is not a product, but a process that needs to be continuously assessed in a periodic manner, as systems evolve and new cyber-threats are discovered. Automation is crucial to tackle this problem. Specially, given the lack of security professionals and how long it takes to train qualified security researchers.

Against the current overwhelming insecurity landscape in robotics, this research paper advocates for offensive security methods for robots. These methods are necessary to understand attackers' behavior, to train defensive mechanisms and ultimately, to help protect existing systems by discovering flaws first. Building upon about a decade of robotics and empowering the use of Artificial Intelligence

(AI) and Game Theory to automate attacks, this line of research aims to study how offensive AI-driven cyber security methods apply to robotics and allow to protect such systems in a feasible manner. In particular, this work presents ExploitFlow (EF), a modular library to produce cyber security exploitation routes (*exploit flows*). ExploitFlow aims to combine and compose exploits from different sources and frameworks, capturing the state of the system being tested in a flow after every discrete action which allows learning attack trees that affect a given system.

ExploitFlow's main motivation is to facilitate and empower Game Theory and Artificial Intelligence (AI) research in cyber security. A secondary motivation is to put ExploitFlow in practice in an area wherein exploitation routes can help significantly secure complex systems via offensive cyber security mechanisms. Robotics is selected as the target area for this research, but other targets might be included in the future. To facilitate adoption, ExploitFlow's syntax and architecture is inspired by TensorFlow[2].

```
1 import exploitflow as ef
2
3 flow = ef.Flow()      # Create a simple exploit flow
4 a = ef.placeholder() # Instantiate a simple operation
5 print(flow.run(a))   # Run the flow and print results
```

---

**Code listing 1** ExploitFlow usage example.

A more complex exploitation route example is depicted below which performs reconnaissance locating multiple targets in the local area network and then conducts scans on each one of them to fill up the state:

```
1 import exploitflow as ef
2
3 flow = ef.Flow()
4 init = ef.Init()
5 recon = ef.Targets() # Build MSF reconnaissance exploit
6 versions = ef.Versions(ports=ef.state.TARGET_PORTS_COMPLETE)
7
8 # initialize state and pass it over a recon action
9 # resulting flow should deliver a state annotated
10 # with the results from the reconnaissance step
11 state = flow.run(init * recon * versions, target="192.168.2.10")
```

---

**Code listing 2** ExploitFlow example doing multi-target reconnaissance. A exploit flow is built that performs reconnaissance locating multiple targets in the local area network and then conducts scans a targeted scan to 192.168.2.10 to fill up a state object that can then be used for reasoning.

To simplify exploitation, ExploitFlow represents each action in an exploitation route with the superclass Exploit, which includes reconnaissance and control actions<sup>1</sup>. Exploits are grouped into six major categories, inspired by the security kill chain[3]. ExploitFlow is a modular, extensible library that accepts connectors for other exploitation frameworks and/or individual exploits, and is composable. ExploitFlow is not an exploitation framework, but a tool to produce cybersecurity exploitation routes, which empowers research in Game Theory and Artificial Intelligence (AI) in cybersecurity.

With this understanding, ExploitFlow aims to contribute to unlocking the potential of modern machine learning approaches and developing a fully automated penetration testing framework that helps produce cybersecurity cognitive engines. Our overall architecture is depicted in Figure 1, which shows our work thus far and our planned contributions for the near future. Our proposed framework, Malism, is designed to enable a user without in-depth security domain knowledge to create their own cybersecurity cognitive engine that helps conduct penetration testing over an extensive range of targets. This framework comprises three primary components:

---

<sup>1</sup>We are well aware that, strictly speaking, reconnaissance scripts don't meet the formal definition of an exploit. However, we still insist on grouping all actions under the same common class (Exploit) to simplify the production of exploitation flows (routes).

1. **ExploitFlow (this paper)**: A modular library to produce cyber security exploitation routes (*exploit flows*). ExploitFlow aims to combine and compose exploits from different sources and frameworks, capturing the state of the system being tested in a flow after every discrete action which allows learning attack trees that affect a given system. ExploitFlow’s main motivation is to facilitate and empower Game Theory and Artificial Intelligence (AI) research in cyber security. It provides a unique representation of the exploitation process that encodes every facet within it. Its representation can be effectively integrated with various penetration testing tools and scripts, such as Metasploit [4] to perform end-to-end penetration testing. Such a representation can be further visualized to guide the human experts for the reproduction of the testing process.
2. **PentestGPT**: An automated penetration testing system that leverages the power of LLMs to produce testing guidance and intuition at every given discrete state. It functions as the core component of the Malism framework, guiding the LLMs to efficiently utilize their domain knowledge in real-world testing scenarios.
3. **PentestPerf**: A comprehensive penetration testing benchmark developed to evaluate the performance of penetration testers and automated tools across a wide array of testing targets. It offers a fair and robust platform for performance comparison.

The harmonious integration of these three components forms an automated, self-evolving penetration testing framework capable of executing penetration tests over various targets, Malism. This framework to develop fully automated penetration testing tools, which we name *cybersecurity cognitive engines*,

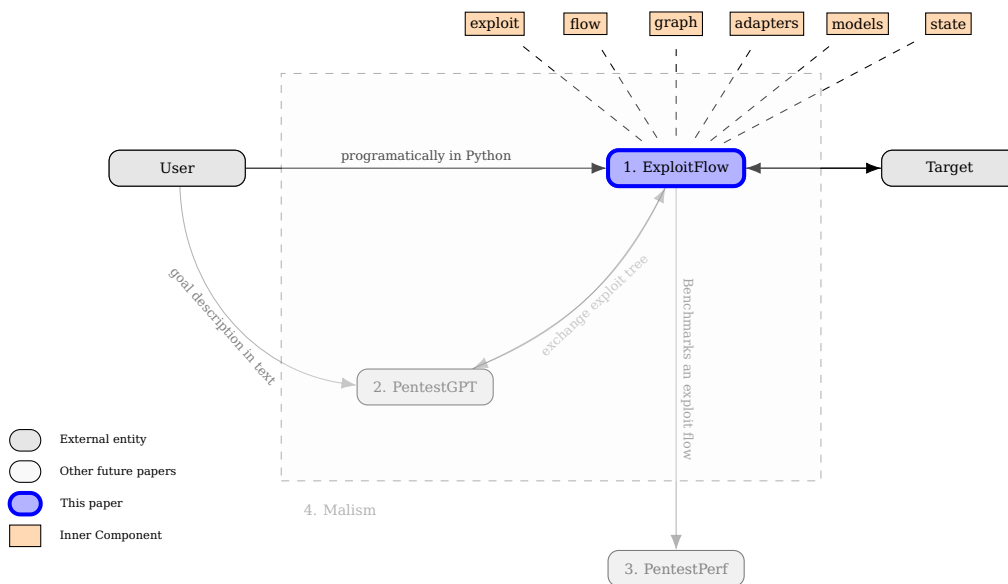


Figure 1: Architecture of our framework to develop a fully automated penetration testing tools, Malism. Figure depicts the various interaction flows that an arbitrary User could follow using Malism to conduct penetration testing (pentest) a given Target. **1. (this paper)** Corresponds with ExploitFlow, a modular library to produce security exploitation routes (*exploit flows*) that captures the state of the system being tested in a flow after every discrete action. **2.** Corresponds with PentestGPT, a testing tool that leverages the power of LLMs to produce testing guidance (heuristics) for every given discrete state. **3.** PentestPerf is a comprehensive penetration testing benchmark to evaluate the performances of penetration testers and automated tools across a wide array of testing targets. **4.** captures Malism, our framework to develop fully automated penetration testing tools which we name *cybersecurity cognitive engines*.

aims to revolutionize the field of penetration testing by significantly reducing the need for domain expertise and enabling more comprehensive and reliable testing.

Contributions of this article are two-fold: first, we present for the first time the implementation and use of ExploitFlow as a tool to build security exploitation routes. Second, we present the results obtained while creating an artificial agent powered by ExploitFlow , that is able to locate exploits affecting the various robotic systems presented to the agent, while minimizing the amount of network traffic and attacks generated (exploits attempted). Q-Learning is used as the learning model and compared against two baselines: a) brute force and b) human expert. The source code including both the ExploitFlow implementation as well as the machine learning experiments are available at <https://github.com/vmayoral/ExploitFlow>.

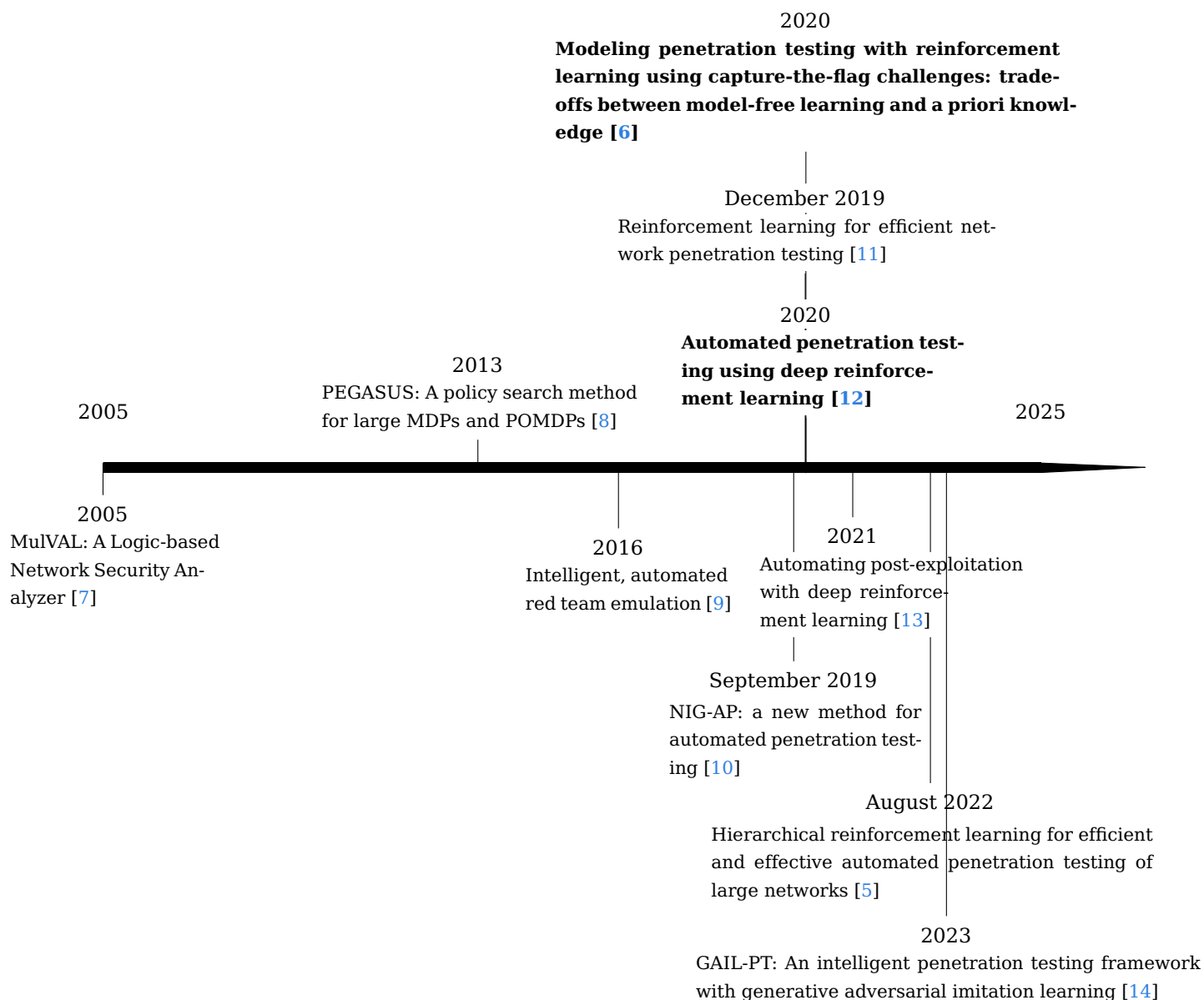
Results presented below in section 3 hint that ExploitFlow is usable and useful for exploring ML use in the context of robot cybersecurity. Three actors are evaluated: the brute-force actor obtains a **-2680** cumulative reward (see 2.3 for an explanation of the reward structure), the human expert **8** and the autonomous agent using value-based Reinforcement Learning (RL) **100**. Attacks graphs shown hint that while the human expert iterates over reconnaissance exploits to introspect the networking scenario, the agent directly learns to submit the exploit that triggers a favourable reward and then iterates over an Idle action to maximize resulting reward (obtaining maximum possible, in fact). This overfitting behavior is a shortcoming to tackle in future work, since it impedes generalization across new testing environments. Overall, ExploitFlow is demonstrated by creating an artificial agent powered by ExploitFlow that is able to locate exploits affecting the target robotic systems presented to the agent, using value-based Reinforcement Learning (Q-Learning ) and while minimizing the amount of network traffic and attacks generated (exploits attempted). ExploitFlow also demonstrates how the state of the system being tested is captured after every discrete action, which allows reasoning, keeping track of action/state pairs, and producing attacks trees affecting a given system. Ultimately, we present in Figure 1 the architecture of our framework to develop a fully automated penetration testing tools, Malism, which we name cybersecurity cognitive engines, and which uses ExploitFlow at its core.

## 2 Methodology

This section describes the research approach, including the data used for the learning task and the learning model. Subsection 2.1 will first set the context, the current state of the art and will highlight the lack of available datasets for robot cybersecurity study. Section 2.2 will then discuss our approach for data generation through CTF-like OS-virtualized environments to generate realistic robotics networking data and what kind of preprocessing and other manipulations were applied to it for the learning task. Finally, Section 2.3 will present an overview of our learning approach and the model used.

## 2.1 Biographical cornerstones in machine learning applied methods to robot cybersecurity

The following presents a summary of some of the most relevant and recent research articles related to offensive cybersecurity approaches that leverage machine learning:



System complexity is the enemy of security. It is very difficult to assure no vulnerabilities in a system that an attacker could exploit. In robotics, the reality is even more concerning, as robots are complex systems, with wide attack surfaces and wherein there is no culture of security yet. As studied in previous work [15, 16], except few vendors and selected actions, no real concern has been shown for security in robotics. Offensive security methods (such as pentesting or red teaming) play an important part in the security lifecycle. In these engagements, as introduced by Applebaum [17], security teams try to break into an organization's assets, identifying vulnerabilities along the way. Red teams take this concept even further, trying to fully emulate what real adversaries do: instead of

just compromising the network and identifying vulnerabilities, they have a larger goal that requires significant post-compromise work.

Various authors, including those summarized above took note of this, and started leveraging offensive mechanisms to tackle the cybersecurity problem. In particular, [18, 19, 20] among others started using machine learning to automate the penetration testing task, presenting different levels of automation. Most of the authors use rather simple neural network architectures, many leveraging Deep Q-Learning (DQN) and mostly involving few hidden layers, besides the input and output ones. This hints about the fact that the tasks learned are rather simple, and a simplification of the complete penetration testing activities. Something coherent across most of these studies is the lack of consistency concerning the representation of the state space for cybersecurity research purposes in ML. Connected to it, we found that most cited authors above use inconsistent datasets. Though various popular *security-oriented networking datasets* exist (see <https://github.com/shramos/Awesome-Cybersecurity-Datasets>) most of the authors reviewed seem to discard these and instead choose to generate their own data via either abstractions (simplified models) or simulated/emulated environments. Further research hinted that rationale behind this is two-fold:

1. Capturing networking data in a scalable manner is non-trivial (and an open problem)
2. Cybersecurity environments are highly non-structured and interactive

In this work, we fail to locate a dataset that could serve the research goals. Deploying offensive cybersecurity approaches in the complex robotics field presents quite a challenge, given its novelty. Also, because the robotics realm is prohibitive: cost, repeatability, and expertise all make it difficult to consistently produce offensive security datasets. Following from this, and similar to some of the cited work above. To proceed with this research, we attempted addressing the two identified hurdles cited above by collecting synthetic data directly from a realistic OS-virtualized scenario with multiple robotic targets. The following sections discuss our approach.

## 2.2 Synthetic data generation through CTF-like OS-virtualized environments

### 2.2.1 Data generation

Obtaining a proper dataset is crucial and specially hard in certain areas, such as the unstructured robotics or cybersecurity. The intersection of both presents an even more difficult challenge. In particular for security, automating offensive cyber security practices constitutes a non-trivial problem because of the range and complexity of actions that a human expert may attempt, which hampers putting together representative datasets. The authors of [6] acknowledge this and focus their attention on simplified penetration testing problems expressed in the form of Capture The Flag (CTF) hacking challenges. They tackle the *dataset production* problem by leveraging existing CTF environments, which they modify to adapt to modern RL *gyms* which produce a reward after an agent performs an action. This way, authors are able to *produce* relevant input data (which serves as datasets) for the learning offensive cyber security practices in selected target scenarios. The research herein will reproduce the approach followed at [6] with relevant extensions meant for modeling robots. The reader should note that while pure networking (for security purposes) modeling can easily be done with virtualization (e.g. OS-virtualization such as Docker, or VMs which are commonly used), robotic environments require a significantly more complex abstraction set, which elevates the complexity of building such CTF environments. To address this problem, the present work builds on top of some of [21], which provides a toolbox for robot cybersecurity concerning. Altogether, we are able to build OS-virtualized CTF environments with the target robots that allow performing RL on them.

Data is thereby generated as synthetic and directly from the simulated robotic environments, each of which can fit a different challenge, or CTF game. The data corresponds to the simulated networking interactions between peers. The following describes how such networking dataflow is abstracted away for learning purposes.

### 2.2.2 Data pre-processing

Learning from computer networking data requires relevant data pre-processing, as the amount of information is overwhelming. The following presents our initial assumptions while considering networking data pre-processing.

Let's assume the following:

- At each time-step, the network is captured by a State class. On its simplest form, each State is represented by a dictionary, each key corresponding with an IP address and holding a sub-state class which captures (for each IP) a) exploits launched against the network and b) port status (open/closed, versions, cpe, etc.).
- One-hot encoding is used as the safest approach to interface with the learning model. Others considered included label encoding and binary encoding. While the latter produced shorter representations, binary encoding might not be suitable for all machine learning algorithms. In binary encoding, although categories are converted into numbers, there is an arbitrary ordering introduced in the categories, which might mislead some algorithms into thinking that there is a relationship between different categories. One-hot encoding instead doesn't have this issue as it uses binary vectors to represent categories, at the cost of increasing the dimensionality of the dataset.

Given these assumptions, we considered initially the following:

- $y = 255$ , number of IPs considered (a complete minor subnet)
- $n = 424$ , number of ports monitored, each represented by  $l = 1$  bit to capture its state (open/close)
- $m = 12$ , number of exploits considered, each represented by  $b = 1$  bits (launched/not-launched), discarding capturing the success/failure of each exploit to model better the uncertainty often encountered in cybersecurity.
- $s = 0$  bits capturing the system information for each IP (OS version, and other meta-data obtained while doing reconnaissance)

These assumptions, which already introduce a significant simplification, while one-hot encoded already lead to the following number of elements while encoding the State:

$$\begin{aligned} y \cdot [n \cdot (n + l) + (m + b) + s] &= \\ 255 \cdot [424 \cdot (424 + 1) + 12 \cdot (12 + 1) + 0] &= \\ 256 \cdot 180,356 &= 46,171,136 \end{aligned}$$

This leads to a setup which is non-feasible computationally. These initial assumptions had to change. We had to significantly increase abstractions (and reduce complexity in terms of network-scope) to make computations tractable in modern general-purpose computers. In particular we used  $y = 7$ ,  $n = 9$ ,  $l = 1$ ,  $m = 12$ ,  $b = 1$  and  $s = 0$  leading to:

$$\begin{aligned} y \cdot [n \cdot (n + l) + (m + b) + s] &= \\ 7 \cdot [9 \cdot (9 + 1) + 12 \cdot (12 + 1) + 0] &= \\ 7 \cdot 246 &= 1,722 \end{aligned}$$

Preserving the modularity principles of ExploitFlow, all the above is implemented by a State class that can easily be extended, overwritten and/or modified per each experiment independently. The listing 3 shows an example implementing a variant of State called State\_v2:



## 2.3 Learning model

As described above, most of the prior art reviewed use rather simple neural network architectures for cybersecurity automation and machine learning, many leveraging Deep Q-Learning (DQN) and mostly involving few hidden layers, besides the input and output ones. In a way, this could be understood as an indicator that the field is still mostly immature (from an ML-perspective) and that simple problems are being explored, which don't demand complex models. In light of this and particularly following the trend established by [6] which focus on scalability, rather than complexity of the problem to solve using a more *explainable* model, we adopt a similar approach and leverage reinforcement learning with simple table-based Q-Learning as a model.

Besides Q-Learning, the following hyperparameters are used consistently across our machine learning efforts: learning rate (alpha,  $\alpha$ ) = 0.1, discount factor (gamma,  $\gamma$ ) = 0.9, exploration factor (epsilon,  $\epsilon$ ) = 0.1.

Finally, the following reward scheme was applied:

- Non-exploits, like idle (no action) and related were assigned a reward of 0, helping the agent to learn that idle might be appropriate given certain networking conditions.
- reconnaissance (both fingerprinting and footprinting) actions were assigned a reward of  $-10$  by default, and an extra  $-1$  for each IP target impacted while incurring in footprinting. This way, we penalize the additional networking traffic generated, which can help Intrusion Detection Systems (IDS) detect malicious activity.
- Formal offensive exploits were assigned a reward of 100 when successful, and  $-100$  if failed.

## 3 Evaluation

Evaluation was performed in a simulated scenario involving the following robotic systems presented in Figure 2. Target objective of the learning effort is to compromise the Universal Robots UR3 collaborative manipulator using [well known security vulnerabilities](#) affecting this robotic system. In particular, an exploit for compromising the robot using the [RVD#672](#) (hard-coded public credentials for controller) vulnerability will be used. Three situations were considered and described below:

### 3.1 Human-expert penetration tester

A human-expert was considered by using ExploitFlow programatically, and manually programming an exploitation route for the best-case scenario. Exploitation route is achieved using the code snippet at listing 5 (complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/9\\_exploitation\\_ur\\_human\\_expert.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/9_exploitation_ur_human_expert.py)).

The execution of this exploitation flow leads to the attack graph depicted in Figure 3 and led to a cumulative reward of 8.

### 3.2 Autonomous agent powered by Q-Learning

An artificial agent powered by ExploitFlow is able to locate exploits affecting the target robotic system, while minimizing the amount of network traffic and attacks generated (exploits attempted) using its table-based Q-Learning model. Exploitation route is learned using the code snippet at listing 6 (complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/11\\_exploitation\\_ur\\_qlearning\\_instances.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/11_exploitation_ur_qlearning_instances.py)).

The execution of this exploitation flow leads to the attack graph depicted in Figure 4 and led to a cumulative reward of 100.



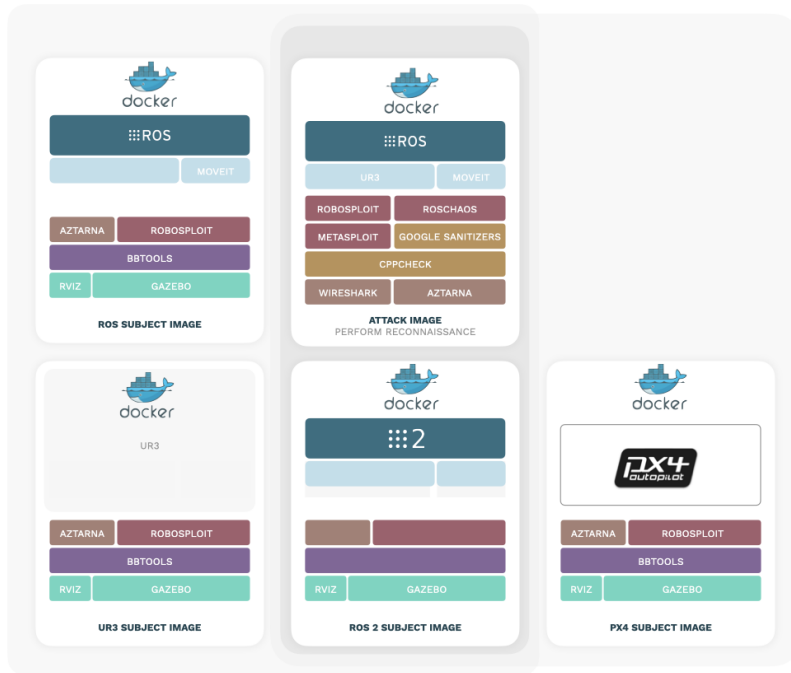


Figure 2: CTF-like OS-virtualized robotic environments involving 4 robotic targets: a robotic brain powered by ROS 2, another powered by ROS, a UR3 cobot manipulator and a PX4 drone. Source code to reproduce this environment is available at <https://github.com/vmayoral/ExploitFlow/blob/main/.devcontainer/docker-compose.yml>

Implementation was performed in Python, as an extension of ExploitFlow (which is coded in Python as well) and in a separate file with standardized interfaces for use. Models abstracted using these interfaces can easily be swapped and tested. Our table-based Q-Learning model presents the form depicted in listing 4. An extension of this implementation to use other models that leverage artificial neural networks (e.g. DQN) is judged trivial if/when leveraging popular machine learning frameworks such as *TensorFlow*.

### 3.3 Brute-force

A brute force baseline (which often the case relates to what many novel pentesters do) is generated by launching all permutations of the selected available exploits against the scenario<sup>2</sup>. Exploitation route is created using the code snippet available at listing 7 (complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/13\\_exploitation\\_ur\\_bruteforce.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/13_exploitation_ur_bruteforce.py)). The execution of this exploitation flow leads to a big attack graph (not shown here) and led to a cumulative reward of  $-2680$ .

## 4 Conclusions and future work

In this article, we made notable contributions to the realm of robot cybersecurity by producing ExploitFlow, a modular library to produce cyber security exploitation routes (*exploit flows*) that allows combining and compose exploits from different sources and frameworks, and captures the state of the system

<sup>2</sup>In fact, a reduced "relevant" set was selected to make it computationally more tractable

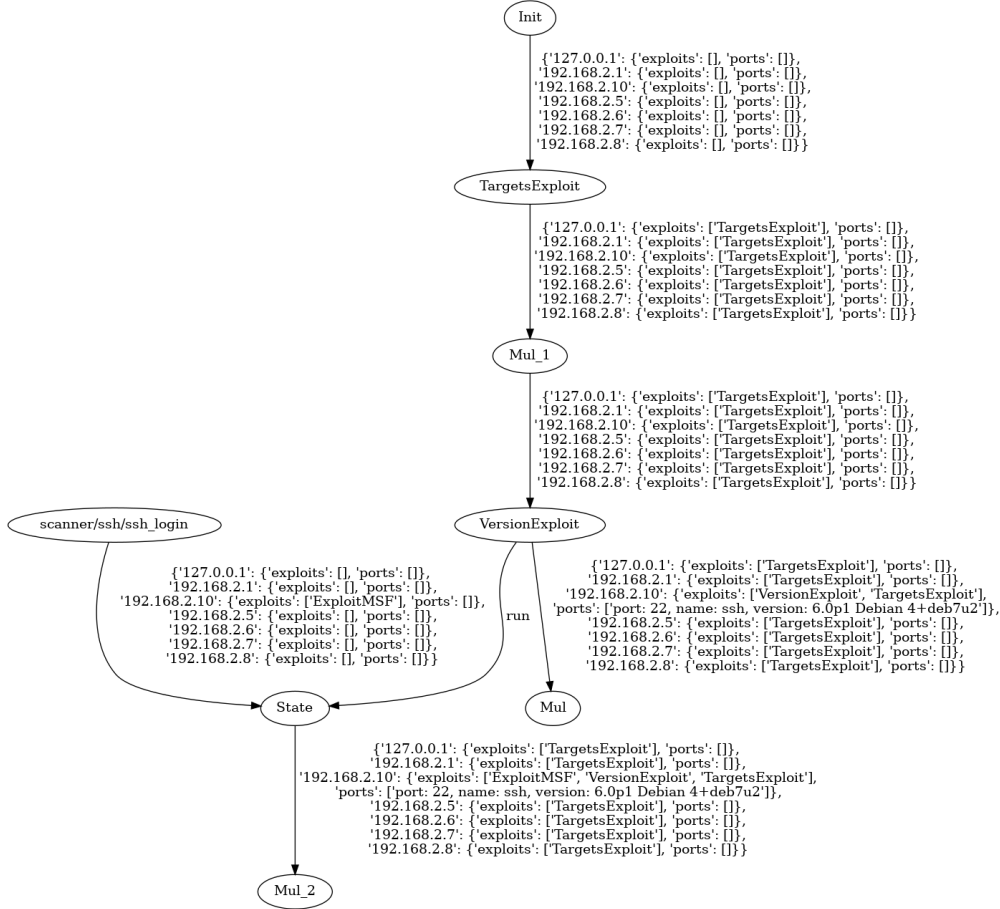


Figure 3: Depiction of the attack graph produced by a human expert while executing the manually-written best-case exploit flow.

being tested in a flow after every discrete action. This allows for further automated reasoning (by means of ML and Game Theory), as well as learning attack trees that affect a given system. Firstly, we introduced the novel implementation of ExploitFlow, an innovative tool devised for crafting security exploitation routes. Furthermore, we successfully demonstrated ExploitFlow by developing an artificial agent with it, that capably identifies exploits in various robotic systems, significantly minimizing both network traffic and the number of exploits attempted.

Using Q-Learning, we compared the efficiency of our model against two baselines: brute force and human expert. Interested readers can delve into the source code and machine learning experiments available at <https://github.com/vmayoral/ExploitFlow>. ExploitFlow’s ability to capture the system state post every discrete action is paramount. It aids in logical reasoning, tracing action/state combinations, and generating attack trees for a specific system. To encapsulate our endeavors, Figure 1 outlines the architecture of Malism, our automated penetration testing tool—termed as a cybersecurity cognitive engine—with ExploitFlow as its foundational element.

Results of the three actors evaluated during this research are summarized in Figure 5, depicting the cumulative reward of the exploit flow generated for each actor, respectively (see 2.3 for an explanation of the reward structure). The brute-force actor obtains a **-2680** cum. reward, the human expert **8** and the autonomous agent using value-based Reinforcement Learning (RL) **100**. While analyzing figures 3 and 4 it becomes apparent that while the human expert iterates over reconnaissance exploits to introspect the networking scenario, the agent directly learns to submit the exploit that triggers a favourable reward and then iterates over an Idle action to maximize the resulting reward (obtaining maximum

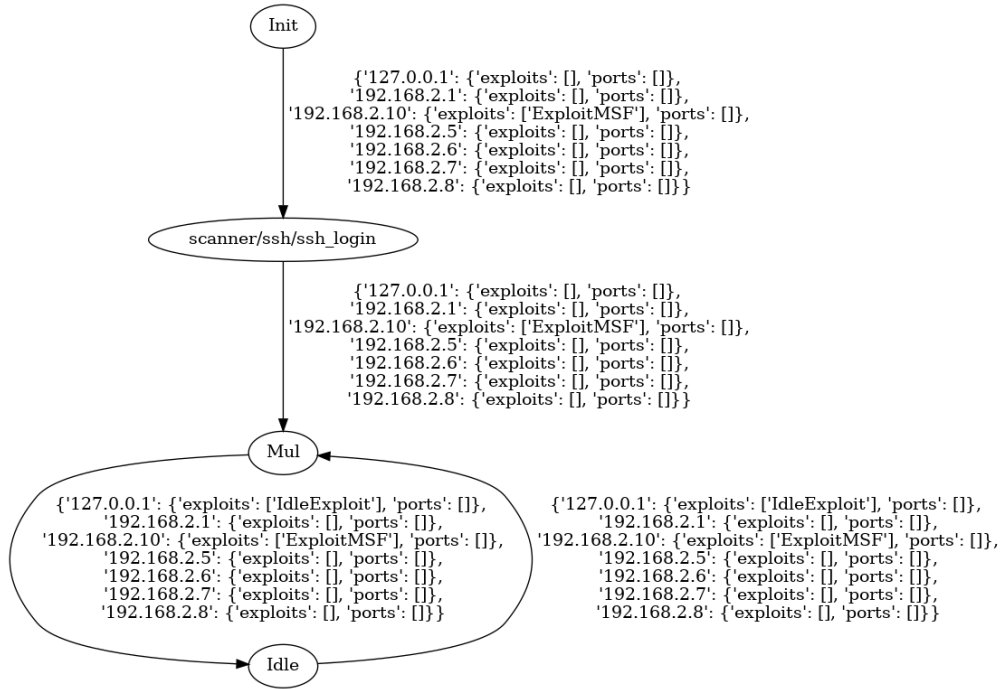


Figure 4: Depiction of the attack graph produced by an autonomous agent powered by Q-Learning algorithm while executing the exploit flow that is derived from its learned value-function (table).

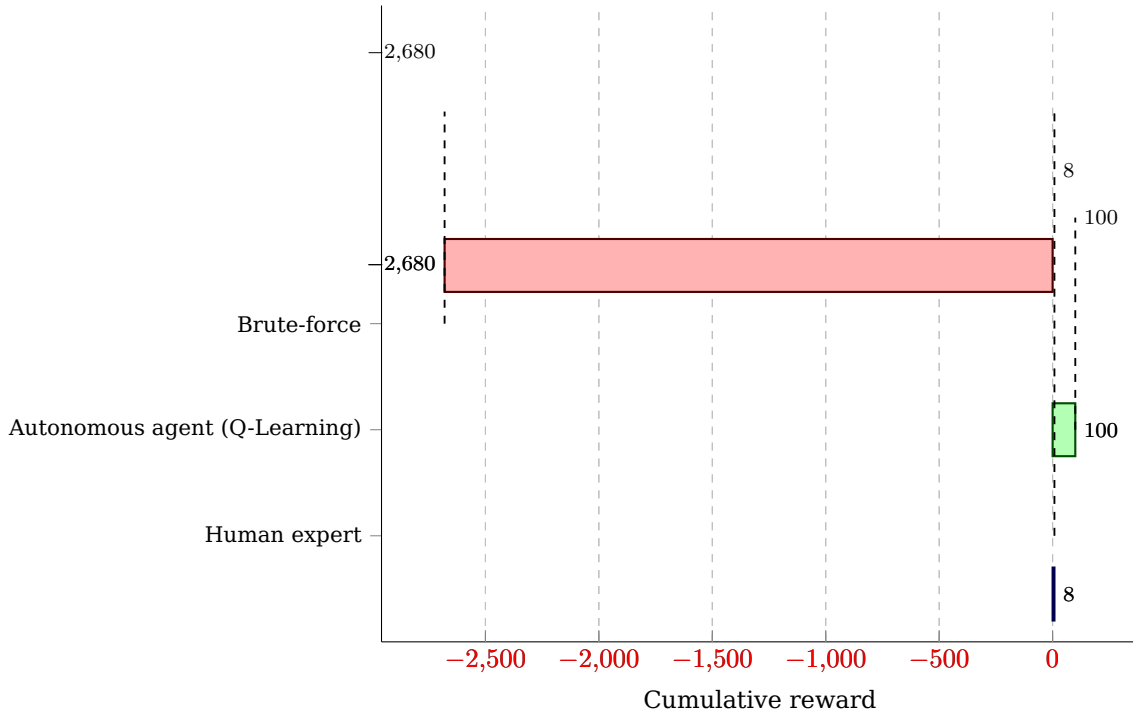


Figure 5: Cumulative reward obtained by three different actors (brute-force, Q-Learning, human expert) while attempting to compromise the target objective in the robot cybersecurity scenario of figure 2

possible, in fact). In light of results, we can conclude that the initial research objectives have been fulfilled. ExploitFlow implementation was shown functional and demonstrated by creating an artificial agent powered by ExploitFlow that is able to locate exploits affecting the target robotic systems presented to the agent, using value-based Reinforcement Learning (Q-Learning) and while minimizing the amount of network traffic and attacks generated (exploits attempted).

Though results meet the initial goals, we must be critical towards identified shortcomings. The following hint a few directions wherein future work could be allocated for further improvements:

- First, current learned functions (value-based) overfit to the particular landscape of exploits available and would perform poorly the moment the networking configuration changes (e.g. IP addresses change). To overcome this, training should be performed without the use of heuristics or simplifications and using *changing* networking setups. Though this can be automated (e.g. creating various CTF-like scenarios), it was beyond the scope of the current effort. A side-effect of this would be the explosion of the state-space, which would probably require to leverage other function representation models, such as artificial neural networks (as opposed to the current table).
- Second, as part of this *overfitting* behavior mentioned, we acknowledge that it is unrealistic for the current autonomous agent to perform properly in the wild, as offensive cybersecurity always requires some level of (continued) reconnaissance. The current agent has not learned to do so. Reward shaping and various scenarios could help addressing this limitation.
- Third, the current one-hot encoding is not scalable (computationally very expensive) and alternatives must be explored if the networking State is to be better captured.
- Fourth, a better representation of the networking state at each timestamp is required to further scale learning. Ideas in this direction include incorporating machine-related metadata (OS, library versions extracted from finger- and foot-printing).
- Fifth and much relevance, there is a huge scalability problem (very costly engineering-wise) while generating datasets that helps autonomous agents learn which action to take next. Creating usable (by ML agents) exploits and scenarios takes a lot of expertise and time. Building appropriate scenarios requires qualified roboticists that understand the dynamics of robotic systems. Adapting exploits for use (or worse, creating new ones) requires cybersecurity experts to craft the right interfaces for automated learning use. To overcome this limitation, modern and popular Large Language Models (LLMs) can be utilized to produce both scenarios as well as exploits in the form/format required, significantly simplifying the effort to produce usable datasets and actions. Listing 8 shows a hint about how LLMs could be leveraged to tackle this shortcoming. We refer interested readers to submodule 2. PentestGPT of Figure 1 for more details on this line of work.
- Sixth, and somewhat connected to the previous one, the heterogeneity of the security networking environments seems to require heuristics to interpret the networking State and take initial actions while tackling new security challenges in unseen environments. This is somewhat connected to the fact that it is unrealistic to capture all know-how concerning offensive attempts, as most is non-disclosed or in a format that can easily be digested and converted into datasets. LLMs could also help on this regard, which is a topic worth researching.

Ongoing efforts to tackle these shortcomings are being developed by the authors. Some available in-the-open and publicly (e.g. at <https://github.com/GreyDGL/PentestGPT>). We also refer interested readers to Figure 1 to get a hint of the overall architecture of Malism, our framework to produce cybersecurity cognitive engines.

## References

- [1] V. Mayoral-Vilches, L. U. S. Juan, U. A. Carbajo, R. Campo, X. S. de Cámara, O. Urzelai, N. García, and E. Gil-Uriarte, "Industrial robot ransomware: Akerbeltz," *arXiv preprint arXiv:1912.07714*, 2019.
- [2] M. Abadi, "Tensorflow: learning functions at scale," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.
- [3] E. Hutchins, M. Cloppert, and R. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Lockheed Martin Corporation White Paper*, 2011. [Online]. Available: <https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf>
- [4] H. D. Moore, "Metasploit framework," 2003, available at: <https://www.metasploit.com>.
- [5] M. C. Ghanem, T. M. Chen, and E. G. Nepomuceno, "Hierarchical reinforcement learning for efficient and effective automated penetration testing of large networks," *Journal of Intelligent Information Systems*, pp. 1–23, 2022.
- [6] F. M. Zennaro and L. Erdodi, "Modeling penetration testing with reinforcement learning using capture-the-flag challenges: trade-offs between model-free learning and a priori knowledge," *arXiv preprint arXiv:2005.12632*, 2020.
- [7] X. Ou, S. Govindavajhala, A. W. Appel *et al.*, "Mulval: A logic-based network security analyzer." in *USENIX security symposium*, vol. 8. Baltimore, MD, 2005, pp. 113–128.
- [8] A. Y. Ng and M. I. Jordan, "Pegasus: A policy search method for large mdps and pomdps," *arXiv preprint arXiv:1301.3878*, 2013.
- [9] A. Applebaum, D. Miller, B. Strom, C. Korban, and R. Wolf, "Intelligent, automated red team emulation," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 363–373.
- [10] T.-y. Zhou, Y.-c. Zang, J.-h. Zhu, and Q.-x. Wang, "Nig-ap: a new method for automated penetration testing," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, no. 9, pp. 1277–1288, 2019.
- [11] M. C. Ghanem and T. M. Chen, "Reinforcement learning for efficient network penetration testing," *Information*, vol. 11, no. 1, p. 6, 2019.
- [12] Z. Hu, R. Beuran, and Y. Tan, "Automated penetration testing using deep reinforcement learning," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 2–10.
- [13] R. Maeda and M. Mimura, "Automating post-exploitation with deep reinforcement learning," *Computers & Security*, vol. 100, p. 102108, 2021.
- [14] J. Chen, S. Hu, H. Zheng, C. Xing, and G. Zhang, "Gail-pt: An intelligent penetration testing framework with generative adversarial imitation learning," *Computers & Security*, vol. 126, p. 103055, 2023.
- [15] L. Alzola Kirschgens, I. Zamalloa Ugarte, E. Gil Uriarte, A. Muñiz Rosas, and V. Mayoral Vilches, "Robot hazards: from safety to security," *ArXiv e-prints*, Jun. 2018.
- [16] V. M. Vilches, L. U. S. Juan, B. Dieber, U. A. Carbajo, and E. Gil-Uriarte, "Introducing the robot vulnerability database (rvd)," *arXiv preprint arXiv:1912.11299*, 2019.

- [17] A. Applebaum, D. Miller, B. Strom, H. Foster, and C. Thomas, "Analysis of automated adversary emulation techniques," in *Proceedings of the Summer Simulation Multi-Conference*. Society for Computer Simulation International, 2017, p. 16.
- [18] M. C. Ghanem and T. M. Chen, "Reinforcement learning for intelligent penetration testing," in *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE, 2018, pp. 185–192.
- [19] S. Niculae, "Applying reinforcement learning and genetic algorithms in game-theoretic cybersecurity," p. 58, 2018.
- [20] J. Schwartz and H. Kurniawati, "Autonomous penetration testing using reinforcement learning," *arXiv preprint arXiv:1905.05965*, 2019.
- [21] V. Mayoral-Vilches, I. Abad-Fernández, M. Pinzger, S. Rass, B. Dieber, A. Cunha, F. J. Rodríguez-Lera, G. Lacava, A. Marotta, F. Martinelli *et al.*, "alurity, a toolbox for robot cybersecurity," *arXiv preprint arXiv:2010.07759*, 2020.

## A Code listings

```
1 class State_v2(State):
2     def __init__(self, *args):
3         self.states = {}
4
5         # initialize all states as empty
6         for ip in TARGET_IP_ADDRESSES:
7             self.add_new(ip)
8
9     def merge(self, newstate, target="127.0.0.1") -> None:
10        """
11        Merges the current object with a new State
12
13        Supports both State_v1 and State_v2.
14        """
15        if type(newstate) == State_v1:
16            self.states[target] = newstate # whether it exists or not
17        elif type(newstate) == State_v2:
18            aux_state = self + newstate # NOTE: overwrites self, with newstate
19            self.states = aux_state.states
20        else:
21            raise TypeError("Unknown state type")
22
23        # (...) various methods omitted
24
25    def one_hot_encode(self):
26        # one-hot encode all State_v1 objects in 'states'
27        states_encoded = [state.one_hot_encode() for state in self.states.values()]
28        flattened_states_encoded = [item for sublist in states_encoded for item in sublist]
29        # return states_encoded
30        return flattened_states_encoded
```


**Code listing 3** Simplified listing of one of ExploitFlow 's State abstractions. Complete source code available at <https://github.com/vmayoral/ExploitFlow/blob/main/exploitflow/state.py>



```

1 class QLearn:
2     """Q-Learning class. Implements the Q-Learning algorithm."""
3
4     def __init__(self,
5                 actions,
6                 epsilon=0.1,
7                 alpha=0.2,
8                 gamma=0.9):
9         self.q = {}
10        self.epsilon = epsilon
11        self.alpha = alpha
12        self.gamma = gamma
13        self.actions = actions
14
15    def learnQ(self, state, action, reward, value, debug=False):
16        """Updates the Q-value for a state-action pair.
17
18        The core Q-Learning update rule.
19         $Q(s, a) += \alpha * (\text{reward}(s,a) + \max(Q(s')) - Q(s,a))$ 
20        """
21        oldv = self.q.get((state, action), None)
22        if oldv is None:
23            self.q[(state, action)] = reward
24        else:
25            self.q[(state, action)] = oldv + self.alpha * (value - oldv)
26
27    def chooseAction(self, state, return_q=False):
28        """An alternative approach for action selection."""
29        # Compute the Q values for each action given the current state
30        q = [self.getQ(state, a) for a in self.actions]
31        maxQ = max(q)
32        if random.random() < self.epsilon:
33            minQ = min(q)
34            mag = max(abs(minQ), abs(maxQ)) # Determine the magnitude
35                                           # range based on minQ and maxQ
36
37            q = [q[i] + random.random() * mag - .5 * mag for i in range(len(self.actions))]
38            maxQ = max(q)
39        count = q.count(maxQ)
40        if count > 1:
41            best = [i for i in range(len(self.actions)) if q[i] == maxQ]
42            i = random.choice(best)
43        else:
44            i = q.index(maxQ)
45        action = self.actions[i]
46        if return_q:
47            return action, q
48        return action
49
50    def learn(self, state1, action1, reward, state2, debug=False):
51        """Get the maximum Q-Value for the next state."""
52        maxqnew = max([self.getQ(state2, a) for a in self.actions])
53        self.learnQ(state1, action1, reward, reward + self.gamma * maxqnew, debug=debug)

```

**Code listing 4** Q-Learning class implementing a common table-based Q-Learning algorithm. Simplified implementation removes comments and non-crucial methods. Complete source code available including comments and documentation at  <https://github.com/vmayoral/ExploitFlow/blob/main/exploitflow/models.py>

```

1 import exploitflow as ef
2 from exploitflow.state import State_v2
3 State_default = State_v2
4
5 flow = ef.Flow()
6 init = ef.Init()
7 recon = ef.Targets()
8 versions = ef.Versions(ports=ef.state.TARGET_PORTS_COMPLETE)
9 state = flow.run(init * recon * versions, target="192.168.2.10")
10
11 for s in state.states.keys():
12     if any((port_state.port == 22 and port_state.open) for port_state in state.states[s].ports):
13         expl = ef.adapter_msf_initializer.get_name("auxiliary", "scanner/ssh/ssh_login")
14         msf_options = {
15             "RHOSTS": s,
16             "USERNAME": "root",
17             "PASSWORD": "easybot"
18         }
19         expl.set_options(msf_options)
20         if not expl.missing():
21             state = flow.run(state * expl, target=s, debug=False)

```

**Code listing 5** Human-expert penetration tester using ExploitFlow programatically programming an exploitation route for the best-case scenario. Complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/9\\_exploitation\\_ur\\_human\\_expert.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/9_exploitation_ur_human_expert.py)

```

1 import exploitflow as ef
2 from exploitflow.state import State_v4
3 State_default = State_v4
4
5 flow = ef.Flow()
6 flow.set_learning_model(ef.QLearn(actions=exploits_encoded, alpha=0.1, gamma=0.9, epsilon=0.1))
7
8 rollouts = 1000
9 episode = 10
10 age = 1
11 debug = False
12 last_10_actions = []
13 while age <= rollouts:
14     if flow.last_state():
15         flow._graph.learning_model.learn(
16             tuple(flow.last_state().one_hot_encode()),
17             flow.last_action().name,
18             flow.last_reward(),
19             tuple(flow.state().one_hot_encode()),
20             debug=False)
21
22     action = flow._graph.learning_model.chooseAction(tuple(flow.state().one_hot_encode()))
23
24     flow.run(flow.state() * action_expl, debug=debug)
25
26     if age % episode == 0:
27         # reset the flow
28         flow.reset()
29
30     # next rollout
31     age += 1

```

**Code listing 6** Code snippet showing the training routine of an autonomous agent powered by Q-Learning. Complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/11\\_exploitation\\_ur\\_qlearning\\_instances.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/11_exploitation_ur_qlearning_instances.py)

```
1 import exploitflow as ef
2 from exploitflow.state import State_v4
3 State_default = State_v4
4
5 flow = ef.Flow()
6 exploits = [ef.idle, ef.metasploit, ef.versions, ef.targets]
7
8 # Get all permutations of the list
9 permutations = list(itertools.permutations(exploits))
10 state = flow.run(ef.init)
11 for perm in permutations:
12     for expl in perm:
13         state = flow.run(flow.state() * expl)
```

**Code listing 7** Code snippet showing the brute-forcing effort of the scenario, trying all possible permutations of the exploits (a reduced set for computational reasons). Complete experiment available at [https://github.com/vmayoral/ExploitFlow/blob/main/examples/13\\_exploitation\\_ur\\_bruteforce.py](https://github.com/vmayoral/ExploitFlow/blob/main/examples/13_exploitation_ur_bruteforce.py)

## B Large Language Models (LLMs) experiments to generate exploits in the desired format

```
1
2 # ChatGPT (GPT-4 model) PROMPT:
3 #####
4 #
5 # Select 10 exploits from the metasploit framework (MSF) and for each, produce an instances of it filling
6 # up the corresponding options required for each exploit while adhering to the following format:
7 #
8 # metasploit_1 = ef.adapter_msf_initializer.get_name("auxiliary", "scanner/ssh/ssh_login")
9 # metasploit_1.name = "scanner/ssh/ssh_login @ 192.168.2.5"
10 # msf_options = {
11 #     "RHOSTS": "192.168.2.5",
12 #     "USERNAME": "root",
13 #     "PASSWORD": "easybot"
14 # }
15 # metasploit_1.set_options(msf_options)
16 # metasploit_1.target = "192.168.2.5"
17 # metasploit_1.reward = -100 # first time successful, will do "*(-1)" to get "+100"
18 #
19 # Provide the code necessary to instantiate the 10 exploits. For each exploit, at the beginning, provide a
20 # small comment of documentation describing what each exploit is expected to do.
21 #####
22 # 1. This auxiliary module will attempt to login to a SSH service with a specific username and password.
23 metasploit_1 = ef.adapter_msf_initializer.get_name("auxiliary", "scanner/ssh/ssh_login")
24 metasploit_1.name = "scanner/ssh/ssh_login @ 192.168.2.5"
25 msf_options_1 = {
26     "RHOSTS": "192.168.2.5",
27     "USERNAME": "root",
28     "PASSWORD": "easybot"
29 }
30 metasploit_1.set_options(msf_options_1)
31 metasploit_1.target = "192.168.2.5"
32 metasploit_1.reward = -100
33 #
34 # 2. This module exploits a buffer overflow in Apache's mod_isapi for versions 2.2.14 and earlier.
35 metasploit_2 = ef.adapter_msf_initializer.get_name("exploit", "windows/http/apache_mod_isapi")
36 metasploit_2.name = "windows/http/apache_mod_isapi @ 192.168.2.6"
37 msf_options_2 = {
38     "RHOSTS": "192.168.2.6",
39     "TARGETURI": "/somepath/"
40 }
41 metasploit_2.set_options(msf_options_2)
42 metasploit_2.target = "192.168.2.6"
43 metasploit_2.reward = -100
44 #
45 # 3. This auxiliary module will scan for vulnerable open FTP servers.
46 metasploit_3 = ef.adapter_msf_initializer.get_name("auxiliary", "scanner/ftp/ftp_version")
47 metasploit_3.name = "scanner/ftp/ftp_version @ 192.168.2.7"
48 msf_options_3 = {
49     "RHOSTS": "192.168.2.7"
50 }
51 metasploit_3.set_options(msf_options_3)
52 metasploit_3.target = "192.168.2.7"
53 metasploit_3.reward = -100
54 #
55 # 4. This module exploits a directory traversal vulnerability in PHPMyAdmin.
56 metasploit_4 = ef.adapter_msf_initializer.get_name("exploit", "multi/http/phpmyadmin_3522_backdoor")
57 metasploit_4.name = "multi/http/phpmyadmin_3522_backdoor @ 192.168.2.8"
58 msf_options_4 = {
59     "RHOSTS": "192.168.2.8",
```

```

59     "TARGETURI": "/phpmyadmin/"
60 }
61 metasploit_4.set_options(msf_options_4)
62 metasploit_4.target = "192.168.2.8"
63 metasploit_4.reward = -100
64
65 # 5. This module exploits a buffer overflow in the Microsoft IIS FTP service.
66 metasploit_5 = ef.adapter_msf_initializer.get_name("exploit", "windows/ftp/ms09_053_ftpd_nlst")
67 metasploit_5.name = "windows/ftp/ms09_053_ftpd_nlst @ 192.168.2.9"
68 msf_options_5 = {
69     "RHOSTS": "192.168.2.9"
70 }
71 metasploit_5.set_options(msf_options_5)
72 metasploit_5.target = "192.168.2.9"
73 metasploit_5.reward = -100
74
75 # 6. This auxiliary module will scan the network for Oracle servers.
76 metasploit_6 = ef.adapter_msf_initializer.get_name("auxiliary", "scanner/oracle/oracle_login")
77 metasploit_6.name = "scanner/oracle/oracle_login @ 192.168.2.10"
78 msf_options_6 = {
79     "RHOSTS": "192.168.2.10"
80 }
81 metasploit_6.set_options(msf_options_6)
82 metasploit_6.target = "192.168.2.10"
83 metasploit_6.reward = -100
84
85 # 7. This module exploits a vulnerability in Microsoft's Server service.
86 metasploit_7 = ef.adapter_msf_initializer.get_name("exploit", "windows/smb/ms08_067_netapi")
87 metasploit_7.name = "windows/smb/ms08_067_netapi @ 192.168.2.11"
88 msf_options_7 = {
89     "RHOSTS": "192.168.2.11"
90 }
91 metasploit_7.set_options(msf_options_7)
92 metasploit_7.target = "192.168.2.11"
93 metasploit_7.reward = -100
94
95 # 8. This module exploits a vulnerability in Windows' handling of VCard files.
96 metasploit_8 = ef.adapter_msf_initializer.get_name("exploit", "windows/fileformat/ms10_026_dbldcode")
97 metasploit_8.name = "windows/fileformat/ms10_026_dbldcode @ 192.168.2.12"
98 msf_options_8 = {
99     "RHOSTS": "192.168.2.12"
100 }
101 metasploit_8.set_options(msf_options_8)
102 metasploit_8.target = "192.168.2.12"
103 metasploit_8.reward = -100
104
105 # 9. This module exploits a buffer overflow in the Samba service.
106 metasploit_9 = ef.adapter_msf_initializer.get_name("exploit", "unix/misc/distcc_exec")
107 metasploit_9.name = "unix/misc/distcc_exec @ 192.168.2.13"
108 msf_options_9 = {
109     "RHOSTS": "192.168.2.13"
110 }
111 metasploit_9.set_options(msf_options_9)
112 metasploit_9.target = "192.168.2.13"
113 metasploit_9.reward = -100
114
115 # 10. This module exploits a vulnerability in the Windows handling of HLP files.
116 metasploit_10 = ef.adapter_msf_initializer.get_name("exploit", "windows/fileformat/
117     ms10_042_helpctr_xss_cmd_exec")
118 metasploit_10.name = "windows/fileformat/ms10_042_helpctr_xss_cmd_exec @ 192.168.2.14"
119 msf_options_10 = {
120     "RHOSTS": "192.168.2.14"
121 }
122 metasploit_10.set_options(msf_options_10)
123 metasploit_10.target = "192.168.2.14"

```

```
123 metasploit_10.reward = -100
```

---

**Code listing 8** Code snippet showing the results of using generative Large Language Models to economically address the scalability problem of generating usable datasets and exploits for the training. The snippet shows how ChatGPT (GPT-4) is able to easily provide exploits for direct use.