Automated Adversary Emulation: A Case for Planning and Acting with Unknowns

Doug Miller, Ron Alford, Andy Applebaum, Henry Foster, Caleb Little, and Blake Strom
The MITRE Corporation
7515 Colshire Drive
McLean, Virginia 22102
{dpmiller, ralford, aapplebaum, hfoster, clittle, bstrom}@mitre.org

Abstract

Adversary emulation assessments offer defenders the ability to view their networks from the point of view of an adversary. Because these assessments are time consuming, there has been recent interest in the automated planning community on using planning to create solutions for an automated adversary to follow. We deviate from existing research, and instead argue that automated adversary emulation – as well as automated penetration testing – should be treated as both a planning *and* an acting problem. Our argument hinges on the fact that adversaries typically have to manage unbounded uncertainty during assessments, which many of the prior techniques do not consider. To illustrate this, we provide examples and a formalism of the problem, and discuss shortcomings in existing planning modeling languages when representing this domain. Additionally, we describe our experiences developing solutions to this problem, including our own custom representation and algorithms. Our work helps characterize the nature of problems in this space, and lays important groundwork for future research.

Introduction

To best understand the security of their systems, network defenders often use *offensive testing* techniques and assessments. These types of assessments come in many forms, ranging from penetration tests – where a team of "white hats" probe the network to identify weaknesses and vulnerabilities – to full-scale red team or even adversary emulation exercises, wherein a team fully emulates an adversary, beginning with reconnaissance, tool and infrastructure development, and initial compromise, and only ending when they reach the specified adversary's goals. As opposed to pure defensive analysis, offensive testing can provide concrete measures of the security of a network by illustrating real attack paths that an adversary could take.

While offensive testing has clear benefits for defenders, it can be difficult for them to actually employ: these tests can be increasingly costly, time-consuming, and personnel constrained. In lieu of easy-to-access offensive testing, an emerging trend in the security community is to launch *automated* offensive assessments. Tools in this space range in capability, from those that focus on technique execution (Smith, Casey 2017) to those that seek to fully emulate an adversary by engaging the full post-compromise adversary life-cycle (Applebaum et al. 2016).

Similarly, the automated planning community has recently taken an interest in security assessments and tests. (Bozic and Wotawa 2017) identify the natural application of automated planning to security: attacks are typically described as a sequence of steps that ultimately achieve a goal, similar in many ways to a plan. They argue that by using automated planning, we can construct tests that we can run against our systems that can identify weaknesses; the authors specifically identify how planning can be used to assess web applications (e.g., SQL injection) as well as the SSL/TLS protocol. Other recent applications include using automated planning and plan recognition to identify larger attack paths (Amos-Binks et al. 2017) as well as vulnerability assessment (Khan and Parkinson 2017).

More specific to offensive testing is the line of work dedicated towards using automated planning specifically for *penetration tests*. Obes, Sarraute, and Richarte (2010) present a model that leverages a deterministic planner alongside a domain description of exploits and connectivity to diagram paths that adversaries could take. Followup work in Sarraute, Buffet, and Hoffmann (2012) expands the model by adding in uncertainty – leaving the core security domain the same – and now using a Partially Observable Markov Decision Process (POMDP) to solve the problem. Shmaryahu et al. (2017) would later acknowledge this POMDP model's success and accuracy, but note its shortcomings – mainly in time-to-compute – as a motivation for using partially observable contingent planning, an approach they argue lies between that of full-knowledge classical planning and multi-belief POMDPs.

Recognizing the wide array of work on automated planning for penetration testing, Hoffman (2015) offers a survey of the literature where he identifies the two main dimensions of existing research: how the approach handles uncertainty from the point of view of the adversary, and how the attack components interact with each other. Hoffman similarly enumerates eight key assumptions, and surveys the literature mapping each to its appropriate assumptions as well as how the approach maps to the two dimensions he identifies.

Despite all of the work dedicated to using automated plan-

ning for penetration testing, little has been done to investigate how the *planning* portion of the problem relates back to the *acting* portion for the problem – most of the approaches assume that the plan will be generated before execution, with the "acting" portion merely following the plan's script. As per Ghallab, Nau, and Traverso (2014), this problem space is not unique in only considering the planning portion of the problem, but for the solutions that have been developed to be commonly deployed, we believe that the field must start embracing acting as part of its paradigm.

Contribution In this paper, we argue that automated adversary emulation – and its cousin, automated penetration testing – is not strictly a *planning* problem, but rather a joint *planning and acting problem*. Our argument hinges on a unique characterization of the uncertainty that adversaries face when targeting a network – specifically, that real adversaries work in the face of *unbounded uncertainty*, wherein they are unable to enumerate the outcomes of a sensing action without actually executing it. This makes it all-but-impossible to create an a-priori plan or policy to account for all states, and, accordingly, adversaries that target systems in the wild tend to interleave planning and acting concurrently.

Our views in this paper are influenced heavily by our prior research in (Applebaum et al. 2016) and (Applebaum et al. 2017), as well as our implementation and testing of the CALDERA automated adversary emulation system[1]. As part of this body of work, we consider the task of adversary emulation as opposed to the traditional penetration testing, and in doing so, the specific techniques we consider in this paper are much more varied than those in the literature which focus exclusively on vulnerabilities and exploits. This paper expands on how the adversary emulation problem should be modeled and describes the integrated planning and acting techniques that we have developed to facilitate automated adversary emulation.

Building Automated Adversary Emulation

The goal of automated adversary emulation is to provide defenders with a tool that is able to execute a full-scale assessment of their network, operating in a way that is similar to a real adversary. Such a tool has significant utility for defenders, including providing a baseline for what their network looks like to an adversary, generating training data, identifying weaknesses and/or misconfigurations, and testing in-place security measures and tools, all the while providing useful empirical evidence for a defensive blue team to build upon. We contrast this with a tool that, for example, only identifies attack paths without executing them: such a tool can provide a map of what the network looks like, but typically will fail to achieve other use cases as it abstracts away important, hard-to-measure details and lacks the realism of actual execution. Specific goals driving automated adversary emulation include:

1. *Intelligent.* The system should choose and chain actions in ways similar to how an adversary would.
2. *Low Overhead.* Defenders should be able to use the tool without needing explicit configuration details, as these are not only time consuming to collect, but are almost impossible for defenders to fully track.
3. *Realism.* The system should execute the same techniques that a real adversary would, and, like a real adversary, should start at initial compromise and only end after achieving (or failing to achieve) a specific set of goals.
4. *Modular.* Users of the system should be able to run assessments with techniques of their choosing, as well as have the ability to add new techniques.

Adversary Model In the context of this paper, we use the MITRE ATT&CK framework[2] as our adversary model, specifically focusing on post-compromise techniques – i.e., those used after an adversary has breached a network – that target enterprise systems. ATT&CK provides added insight into the adversary's lifecycle by decomposing it into the top-level tactical goals that adversaries try to achieve and the techniques that adversaries use to achieve those goals. ATT&CK is unlike other threat models in that it was built by analyzing publicly available threat reports – each technique in ATT&CK is grounded in that it has either been used actively by real advanced persistent threats, or that it is common knowledge for red teamers. Moreover, whereas other threat models tend to overly focus on vulnerabilities and exploits, ATT&CK describes behaviors commonly employed by real adversaries, which increasingly involve re-using benign, normal functionality (e.g., built-in system tools) to achieve malicious effects. These features position ATT&CK well within the context of our goals.

Characterizing Uncertainty in Automated Adversary Emulation

Cyber intrusions can be broken down into a series of constituent actions executed by the adversary. These actions typically fit into two buckets: actions that expand the adversary's foothold, and actions that expand on what the adversary knows. Depending on the circumstances, some actions can span both categories by expanding on the adversary's foothold while also providing new knowledge. To illustrate this, below we describe three common actions – taken from the ATT&CK framework – that adversaries typically execute during engagements.

Exploiting a Vulnerability When most people think of cyber attacks, they think of zero-days and exploits used against vulnerable software. With this technique, the adversary expands its foothold by exploiting a vulnerability – i.e., a buffer overflow, remote code inclusion, SQL injection, etc. – on a target system in order to gain access or achieve a malicious effect. Adversaries can have a variety of end goals when using this technique, though common ones include exploiting a remote system for lateral movement and exploiting a local kernel vulnerability for privilege escalation. To successfully launch this technique, an adversary

[1] https://github.com/mitre/caldera

[2] attack.mitre.org

need only have access to the knowledge the vulnerability exists, the right exploit code, and know that the vulnerability is present on the target. This technique expands the adversary's foothold and is an action the adversary takes to acquire new access which can be a new system or level of privilege.

Remote System Discovery In order for an adversary to operate against a network, they need to know exactly what systems are on that network. Consider the case for an adversary who just successfully phished an employee within an enterprise; after enumerating the details of the compromised host, they would likely try to seek out another host within the internal network so that they can enlarge their foothold. Before expanding laterally, the adversary would need to *discover* the remote systems first. Depending on the platform, there are many ways to achieve this (e.g., `ping`) – in Windows enterprise systems, the common way is running the `net view` command, which will return a list of all hosts in the local domain. This technique is an example of a *knowledge gaining* technique.

Credential Dumping Credential dumping is a favorite for red teamers and real adversaries targeting enterprise systems. To run this technique, an adversary only needs elevated (i.e., SYSTEM) access on a target host. After running it, the technique will extract all cached domain credentials (i.e., passwords and/or hashed passwords) from the running host; cached credentials include all of the credentials of the accounts of users that have logged on since the last reboot. Extracted credentials can be useful later for the adversary as they laterally move through the network, using the stolen credentials to access capabilities they would not otherwise be able to access. This technique both gains new knowledge (e.g., what accounts exist) as well as expands the adversary's territory (i.e., by gaining access to credentials).

Central to each three of these actions is the concept of uncertainty. While not covered here, we note that some actions can also create new domain objects (for example, remotely copying a file from a compromised host to an uncompromised host as a means for lateral movement). This can pose an interesting problem for planning, as many representations do not allow for the creation of new domain objects.

Managing Uncertainty

Uncertainty factors into each of these techniques in vastly different ways. Consider the first technique, exploiting a vulnerability. In executing this technique, we can characterize two main sources of uncertainty:

- Is the target susceptible to this exploit?
- Was the exploit technique executed successfully?

Both of these questions are *enumerable*: each is a simple yes or no question, making it easy to construct contingency plans that branch over all scenarios. Additionally, these outcomes are *sensible*, in that we can execute other techniques that e.g. check to see if a target is susceptible to an exploit or determine if an exploit ran successfully. Indeed, the approach in (Shmaryahu et al. 2017) explicitly models sensing actions for both. After successfully executing this technique, the adversary will have a new foothold (in the case of lateral movement) or have elevated privileges on an already compromised host (in the case of privilege escalation).

The uncertainty when dumping credentials, however, is different. Unlike exploiting a vulnerability, the uncertainty in dumping credentials is specifically in the *technique output*. When dumping credentials, we know that the adversary will obtain all cached credentials, but in practice the adversary will rarely have any apriori knowledge of what these cached credentials are. This makes it much harder to encode than exploiting a vulnerability, as the adversary may get:

- no credentials;
- credentials for accounts it has never heard of;
- credentials that it can not currently use; or
- credentials that it can immediately use.

In fact, these outcomes tend to occur together: the adversary will likely obtain credentials for accounts it has not heard of while also obtaining credentials for accounts that it has heard of. Enumerating each of these states is possible at the abstract level, but this approach decouples the action from the grounded solution – i.e., references to the objects in the environment, such as John's account or Pete's workstation – making it hard to link the consequences of dumping credentials to enabling actions in the future, and further making it difficult to do goal-based planning to achieve real objectives.

At a more abstract level, adversaries tend to execute techniques that deal with *unbounded uncertainty* when targeting systems; while some of the uncertainty can be characterized, the uncertainty is typically hard to qualify in a way that can be explicitly *bound* without losing too much precision. We contrast this description with *bound uncertainty*. To understand the distinction, consider an action that scans a target for running services to discover exploits. We might consider such an action as being able to identify vulnerability 1, vulnerability 2, ..., etc., where the quantity of vulnerabilities is a known, finite amount that reflects the adversary's toolkit. Scanning for vulnerabilities, then, is a mapping from the unknown state into one where zero or more vulnerabilities – that have already been enumerated beforehand – are known. By contrast, dumping credentials can result in any number of real accounts being discovered, and while each individual account may have some sort of mapping, it is hard for the adversary to explicitly bound the uncertainty, as it does not know the accounts that it does not know.

This kind of scenario is commonplace, and while adversaries will target networks with *specific* goals in mind, they will often bring about those goals in *non-specific* ways. Adversaries typically approach networks with a mental playbook – based on their goals, experience, and results – specifying how they should generally behave, describing their tactical goals as well as the constituent actions they should use throughout the intrusion. How these actions are ordered is left to the adversary to determine at run-time: because of the extreme uncertainty that adversaries have when operating against networks, conformant or contingent plans are too difficult for an adversary to construct when operating in a network. Nonetheless, as we seek to automate the adversary

emulation process, it is critical that we attempt to provide as much detail – and intelligence – that we can in order to have the most accurate results.

Formal Problem Description

We define our planning problem in two stages: first, the large, agent-agnostic description of the problem, and the smaller adversary-oriented view of what the problem looks like. For the former, we define the general description of the problem as a quadruple: $\pi = \langle \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{P}$ is a set of propositions, $\mathcal{A}$ a set of actions, $\mathcal{I} \subset P$ the starting state (i.e., a set of propositions), and $\mathcal{G} \subset P$ the goal propositions. For every proposition $p \in \mathcal{P}$, p can be assigned a truth value of either $true$ or $false$. Each action $a \in \mathcal{A}$ is a double, $\{pre_a, post_a\}$, where pre_a is the set of propositions that must be true to execute action a (i.e, the preconditions), and $post_a$ is the set of propositions that will be true after executing a. In our model, we assume that the truth of $\mathcal{P}$ is static – i.e., a proposition p's truth value remains the same unless changed by the adversary.

A solution to π is a sequence of actions $S = \{a_1, a_2, ..., a_n\}$ such that, when started from $\mathcal{I}$, executing the actions of S in sequence would result in the propositions in $\mathcal{G}$ all being true (or false, if specified as such). We refer to this as an *conformant* solution.

In our scenario, a conformant solution to π is unrealistic due to the adversary's uncertainty: because the agent has *unbounded uncertainty*, it is unlikely for our adversary to be able to construct an explicit solution before acting. Thus, we redefine our problem as follows: let π_i be a tuple $\pi_i = \langle \mathcal{F}_i, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}_i \subseteq \mathcal{P}$ is the set of propositions that the adversary *knows exist* at time i, regardless of whether the adversary knows their truth value. At first glance, it might seem that if a proposition $p \in \mathcal{F}_i$, then the adversary knows the truth value of p, however this is not always the case. As an example, an adversary may dump credentials and find that joe is a user account. Because joe is a user account, the adversary can infer that joe may be a domain admin – i.e., $domain(joe) \in \mathcal{F}_i$ – but while the adversary can infer this proposition exists, it does not know if it is true.

The differentiator between π and π_i is in the space of propositions – an actor working with π has a fully enumerated proposition space, while an actor working with π_i knows that the proposition space is only a subset of what truly exists. Because of this, solving π_i is different than solving π in that the former *necessitates* acting: the agent *must* perform discovery actions to identify unknown propositions, with planning beyond this point particularly difficult. We can contrast that with an agent working with π who knows that there are no new propositions to be gained during an operation, and thus can plan for all contingencies. Under this formalism, then, the adversary emulation problem should not be treated as a strict *planning* problem, but rather as a *selection* problem, where the agent seeks to find the best action *now*, to maximize its chances of achieving $\mathcal{G}$ in the future.

Representing Planning Problems for Adversary Emulation

The formalism of π_i in the previous section calls for the representation of a set of propositions to denote the cyber domain, but is agnostic as to the particular language that represents the propositions. At first glance, this may seem to be a relatively unimportant detail, but our experiments have shown that this situation mandates nuance when represented as a planning problem. Generally, we have observed the following guidelines when tackling this problem:

Object-oriented description. Our representations allow us to reason about the objects (i.e., hosts, users, accounts, etc.) typically found in networks in contrast to e.g. state-based approaches. This approach offers many benefits, and is particularly relevant as objects in the cyber domain are well-structured; for example, networks are typically comprised of multiple hosts, each of which have standardized fields such as fully qualified domain name, user accounts that are administrators, operating system, etc. More abstractly, this guideline lends itself well towards *object-oriented planning* (Katz, Moshkovich, and Karpas 2016).

Parameterized actions. We tend to refer to our action space as a set of *ungrounded, parameterized* actions; colloquially, we might traditionally refer to these as *techniques*. This, in large part, is due to the uncertainty problem mentioned before: suppose we have a *technique* of dumping credentials from a host. In a traditional representation, we would have n instances of this action represented in $\mathcal{A}$, where n is the number of hosts in the network; i.e., dumping credentials on host 1 is an action, dumping credentials on host 2 is an action, etc. However, because the adversary does not know all of the hosts on the network, it might only have access to a subset of $\mathcal{A}$ alongside the ungrounded, parameterized version of the action.

Deterministic action outcome. Executing an action will always result in the same outcome in the same environment. This has likewise been acknowledged in the POMDP approach (Sarraute, Buffet, and Hoffmann 2012), where uncertainty is instead modeled in the adversary's belief space of what the current state is (which indeed more accurately represents what penetration testers do in practice). Alternative approaches, such as the one in (Durkota 2014), abstract this uncertainty instead into the action's outcome.

Monotonic action consequences. Our representations are all delete-free. This assumption appears to be consistent with the literature – no representations that we have seen explicitly model deletes – although we note that the solution techniques such as using POMDPs (Sarraute, Buffet, and Hoffmann 2012) or contingency planning (Shmaryahu et al. 2017) are robust enough to handle deletes if the model does include them.

Early Work in $\mathcal{K}$

Our first attempt (Applebaum et al. 2016), (Applebaum et al. 2017) at modeling this problem was to use a representation encoded in the $\mathcal{K}$ (Eiter et al. 2000) planning language,

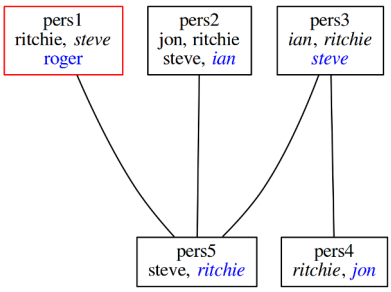

Figure 1: Example encoding of an enterprise system. Each box represents an individual workstation with edges representing allowed traffic flows. The name of the workstation is on the first line within each box, with authorized remote logins in black and local administrators in blue. Active logins are denoted by italics. The red box around $pers1$ signifies the adversary's foothold there.

which we could solve using the DLV^{K}[3] planning system. We initially chose $\mathcal{K}$ as the implementation language due to its ability to express uncertainty, our familiarity in Datalog, the ability to encode inferences and axioms, and the availability of the DLV^{K} solver, which itself gave us a fair degree of flexibility during use.

The initial data model that we constructed was simple: it only contained two types of objects, accounts and hosts. Our fluents described both the state of the network – including hosts that were connected and which accounts could log in where – as well as the state of the adversary. This introduced our first challenge, in that we could have a fluent which described the state of the world, and then we would need a corresponding fluent to denote when the adversary was aware of that state. As an example:

```
connected(X, Y) requires host(X), host(Y).
knowsConnected(X, Y) requires host(X), host(Y).
```

Above, the first first predicate `connected(X, Y)` denotes that two hosts in the model can communicate over the network and the second adds it to the adversary's knowledge base. To discover these connections, the adversary has access to a simple action:

```
executable enumerateHost(X) if hasFoothold(X),
   escalated(X), not hostEnumerated(X).
caused knowsConnected(X, Y) if connected(X, Y)
   after enumerateHost(X).
caused hostEnumerated(X) after enumerateHost(X).
```

In words, for the adversary to execute the `enumerateHost` action, it must have an escalated foothold (i.e., executing under root or SYSTEM) on a host and have not previously enumerated it. After execution, it will know all valid connections to or from that host.

Using this problem encoding, we were able to construct plans over our model for the adversary to achieve arbitrary goals. As an example walkthrough, consider the network represented in Figure 1. From this view, we can see a clear path from the adversary's initial foothold on $pers1$ to reach $pers4$ with the following plan: dump credentials on $pers1$ to obtain $steve$'s credentials, use $steve$ to remotely log in to $pers5$, use $steve$ again to move from $pers5$ to $pers3$, dump credentials on $pers3$ to obtain $ritchie$'s credentials, and then use $ritchie$'s account to remotely login to $pers4$[4].

This representation is useful in mapping out the weaknesses in the generated networks from a general perspective, but stops short of being able to entirely represent the features needed for automated adversary emulation: from the adversary's point of view, the adversary only has *partial* view of the network and cannot deterministically reason about the consequences of actions. In Figure 1, the adversary only has a foothold on $pers1$, and without doing anything, only knows that $pers1$ exists, nevermind any of the accounts it would need to laterally move to $pers4$ (or even that $pers4$ exists). Consider the action to dump credentials:

```
executable dumpCreds(X) if hasFoothold(X), escalated(X).
caused knowsCreds(A) if activeCreds(A, X)
    after dumpCreds(X).
```

It is easy to see when the adversary can dump credentials: it only needs an escalated foothold on a host to do so. However, the exact consequences to the adversary are unknown and unbounded: this predicate can only be evaluated *after* running this technique, as the active credentials on a host are unmeasurable from the adversary's point of view.

Developing a Planning and Acting Environment To facilitate experiments with the DLV^{K} format, we developed a turn-based simulation system wherein an adversary agent could maintain its own internal state, interfacing with a global agent that had full visibility of the world. Our adversary agent starts with a simple view of the world – it has an initial foothold on the network – as well as the action definitions and inference rules that are used in the real model. It does not, however, know anything about the network: it is unaware of what hosts and accounts exist, what the topology looks like, what the trust relationships are, etc. Instead, the adversary learns these features as it executes actions.

During a simulation run, the adversary sends its chosen action to the global agent, which first checks to see if the action is legal, and, if so, determines what changes should be made to the real model and which new knowledge should be passed to the adversary. As an example, looking at Figure 1, the adversary would gain `knowsRemote(steve, pers1)` and `knowsCreds(steve)` after executing `dumpCreds(pers1)`.

Pythonic Representation

Our work using $\mathcal{K}$ was useful as a testbed for us to develop planning algorithms. However, from an implementation standpoint, it had several weaknesses, primarily in converting from it to what our tool was executing and what it needed for technique execution; $\mathcal{K}$ did not follow the object-oriented guideline we would ideally follow. This led us to develop a custom representation that easily facilitated both planning and acting.

[3] http://www.dlvsystem.com/k-planning-system/

[4] Note that in our model the adversary would have to perform some knowledge gathering actions throughout this plan as well.

Actions in CALDERA are each represented as a Python class. Each class contains fields that describe how the technique is executed and implemented, some metadata, and then a Python representation of the logic. At a high level, the logic provides information on the pre and postconditions of the actions, represented in a way that talks strictly about the objects that are involved[5]. Treating pre and postconditions as restrictions on objects meshed well with our implementation: the logic explicitly maps to the objects in the database schema and couples well with the action execution code. The internal data model features 15 top level objects, each of which has a set of fields. Field values can either be integers, strings, references to other objects, lists of things, booleans, or dates. Example objects and fields include: Remote Access Trojans (RATs), which have host (object), elevated (boolean), and username (string) fields, and Hosts, which have admins (list), fully qualified domain name (string), and hostname (string) fields, amongst others.

One of the downsides of the Pythonic representation is difficulty for human operators to read. For example, consider the following action to copy a file to a network share:

```
preconditions = [("rat", OPRat),
  ("share", OPShare({"src_host": OPVar("rat.host")}))]
postconditions =
  [("file_g", OPFile({'host': OPVar("share.dest_host")}))]
preproperties = ['rat.executable', 'share.share_path']
postproperties = ['file_g.path']
```

This syntax defines four main components: preconditions, which *must* be true to execute the technique, postconditions, which *at least* will be true after executing the technique, pre-properties, which are things that must be *defined* to execute the technique, and post-properties, which are things that will be defined after executing the technique. Parsing each of these, the first precondition states the the identifier `rat` must be of type `OPRat`. The second requirement states that `share` must be of type `OPShare`, where the `src_host` field is equal to the `rat`'s `host` field. The postcondition states that a new object `file_g` of type `OPFile` will be created, where the `host` field of the new `file_g` object is equal to the original `share`'s `dest_host` field. The pre-properties specify the `rat`'s `executable` and the `share`'s `share_path` fields must also be defined, and the post-properties state that the `file_g`'s `path` field will be defined after execution.

This representation facilitates reasoning over *objects* as opposed to strictly properties, and is handy in the cyber domain: most of the techniques either add knowledge or create objects, with modifying objects an atypical use case. In fact, both adding knowledge and creating objects are represented the same in the Pythonic representation – both involve the agent adding new objects (either those that are discovered or those that are created) to its knowledge base.

Converting the Pythonic Representation Instead of reasoning directly with the Pythonic representation, we created an intermediary language that converted the Python requirements to Datalog[6]. For example, the copy action above was converted as follows:

Figure 2: Workflow of plan-and-act paradigms considered for automated adversary emulation.

```
Parameters:
      EXECUTABLE, HOST, RAT, SHARE, SHARE_PATH, SRC_HOST
Preconditions:
      has_property(RAT, executable, EXECUTABLE)
      has_property(RAT, host, SRC_HOST)
      has_property(SHARE, dest_host, HOST)
      has_property(SHARE, share_path, SHARE_PATH)
      has_property(SHARE, src_host, SRC_HOST)
      oprat(RAT)
      opshare(SHARE)
Postconditions:
    + defines_property(FILE_G, path)
    + has_property(FILE_G, host, HOST)
    + opfile(FILE_G)
```

This conversion is relatively straightforward: we see predicates like `oprat(RAT)`, which declare that the `RAT` parameter must be of type `oprat`, matching the Pythonic requirement. Under the preconditions, the second and fifth predicates mandate that the `host` property of `RAT` must be the same as the `src_host` property of `SHARE`. For preproperties, note that the first precondition – `has_property(RAT, executable, EXECUTABLE)` – is the only one to specify a requirement on `EXECUTABLE`; this parameter must merely be defined, but does not need to be explicit.

While the Pythonic code is dense, its Datalog translation is very straightforward. Each object requirement is specified as a type restriction in Datalog. Each preproperty or postproperty is specified as an unbounded `has_property` statement on that object. Preconditions and postconditions are specified with type requirements as well as specific restrictions over properties, again leveraging the `has_property` predicate.

Choosing Adversary Techniques

In this section, we discuss some of the practical solutions that we have experimented with to solve the planning and acting problem for automated adversary emulation, describing theoretical algorithms that work in our $\mathcal{K}$ environment as

[5]A full description of the syntax can be found at `http://caldera.readthedocs.io/en/latest/add_technique.html`.

[6]For ease of integration, we avoided converting to $\mathcal{K}$ – which required the DLV solver – and instead converted to native Datalog.

well as the one implemented in CALDERA. All of the algorithms discussed in this section leverage the same planning-and-acting paradigm (visualized in Figure 2):

1. Obtain and update the world state.
2. With the world state, use the precondition model to identify which actions are valid at the current time step.
3. Building off of step 2, construct a set of plausible plans.
4. Evaluate each plan constructed in the previous step.
5. Execute the first action in the highest rated plan.
6. Observe the responses, stopping if the goal state has been observed and going back to step 1 otherwise.

This paradigm operates in a way that is *fire-and-forget*: even though the algorithms construct plans, they only execute the first action in the plan, completely re-planning each time they have to construct new plans.

Evaluating Plans

All of our techniques evaluate plans based on the metric first proposed in (Applebaum et al. 2016). This technique assumes we have access to some reward function $R : \mathcal{A} \rightarrow \mathbb{R}$ that maps each action to some numeric reward. Then, given a set P of plans, where each plan is a sequence of actions $a_1, ..., a_n$: for a plan $p \in P$, we define its score as:

$$S(p) = \sum_{i=1}^{n} \frac{R(a_i)}{i}$$

In words, each plan is assigned a score that represents a decreasing weighted sum over its constituent actions. We note that this is in fact very similar to using a finite horizon over a Markov decision process (MDP) to calculate reward, although here we use a linearly decreasing score as opposed to an exponential one that is typically used in MDPs. In line with the guidelines that we discussed in the previous section, the scoring algorithm treats each action as ungrounded – dumping credentials, for example, on host 1 would yield the same reward as dumping credentials on host 2, regardless of any differences in hosts 1 and 2.

Constructing Plans

The difficulty in constructing real plans stems primarily from the knowledge disparity facing the adversary: there are propositions in the world-space that the adversary is unaware of. In the generalized case, reasoning over all unknown propositions would be challenging, however, by leveraging our representation – i.e., that we have a data schema, ungrounded action definitions, and a feel for what the world should look like – we can still approximate what might be considered "good" solutions to this problem. Thus, our initial approach worked as follows:

1. Construct a fictional world $\mathcal{P}'$.
2. Merge $\mathcal{P}'$ with $\mathcal{F}_i$. In the case that some proposition in $\mathcal{P}'$ conflicts with $\mathcal{F}_i$, defer to the known proposition to ensure consistency.
3. Initialize an empty set of plans, P.
4. For each action type – i.e., for each ungrounded action – obtain the set of plans that executes that action the soonest. For example, if we can dump credentials – regardless of the host – at time step three and no sooner, add all plans of length three that dump credentials to P. Repeat this for each action type.
5. Return P.

This process is executed multiple times each time step in a Monte-Carlo style simulation: after each individual run, P would be evaluated and the best action would be recorded. After running all of the Monte-Carlo trials, each "best" action would be given a vote, and the action with the most votes would be executed. Experimenting in our $\mathcal{K}$ domain, this setup performed reasonably well, outperforming strategies that iterated through actions in a discrete sequence (i.e., a finite-state machine) as well as a greedy strategy that skipped the plan construction step.

In practice, however, this technique was unfeasible as it required *full world* simulation when constructing $\mathcal{P}'$. This is largely impractical as the more the planner needs to "guess" what the real world looks like, the more inaccuracies it will add. Moreover, creating the entire network and reasoning over it was a time consuming procedure.

To improve on this procedure, we created a variant of the above approach that differed only in how it constructed the initial $\mathcal{P}'$ fictional world. Instead of trying to fully simulate what the world might look like, the planner would make small increments to $\mathcal{F}_i$ based on a fixed set of rules that would enumerate some of the potential configurations. For example, if the planner knew some hosts existed, but did not know the admins on those hosts, the planner would guess who the admins were, with probabilities for guessing that a known or unknown account was an admin. This slight modification provided significant performance increases over our initial approach; the full results of these tests can be found in (Applebaum et al. 2017).

While these two approaches provided strong laboratory-based results, they both sat too far away from the actual CALDERA implementation. Both approaches proved to be too intensive for the large data model that CALDERA was using – how do we, for example, simulate what a random process might look like, given that it has 15 different fields? How many processes should we infer exist? Moreover, the representation in $\mathcal{K}$ did not lend itself well to meeting our guidelines for modeling this domain.

Instead, we developed a relatively simplistic approach that leveraged our Pythonic data representation, converted to Datalog. To get plans, the planner would explore each possible action – in sequence – popping its execution onto a stack and recursing, stopping when it reaches a fixed depth; in essence, the planner explores all possible plans of a fixed length starting at the current state via a depth-first search over the action space. This approach is fairly immature as opposed to traditional planning techniques, however we developed several heuristics to help reduce redundancy and optimize the search:

- The algorithm never explores the same action twice.
- If two actions have the same effects, only explore one.

To understand this, consider the following action to identify all hosts running in a domain:

```
Parameters:
        RAT
Preconditions:
        oprat(RAT)
Postconditions:
      + defines_property(HOST_G, fqdn)
      + defines_property(HOST_G, os_version)
      + ophost(HOST_G)
      + oposversion(OS_VERSION_G)
```

To check if this action is valid, the planner runs a Datalog query to ground its preconditions, in this case returning all objects of type `oprat`. After execution, it will create a new host – "`+ ophost(HOST_G)`" – with the associated properties; from an execution standpoint, the planner, when considering this action, will *internally* execute it, adding the appropriate facts to the knowledge base. In this case, since the postconditions are not tied to the preconditions or parameters, all of the facts will be brand new – that is, it will know that it needs to create new objects to match the conditions. Once it finishes exploring this path, it will pop these postconditions off its stack and move on to the next branch in the tree. To avoid complexity, if multiple RATs match the precondition, it will only explore the paths with one of them since the postconditions are the same, regardless of the RAT. Note that instead of explicitly declaring its in-practice functionality – discovering all hosts – the representation only adds *one* new host to the knowledge base.

Interestingly, this approach completely eschews the need to simulate or guess what the unknown propositions in the world are by leveraging its representation. While we have not conducted any rigorous trials to showcase its efficacy, we have found that, with this algorithm and representation, CALDERA is able to successfully achieve full compromise of setup lab environments. Prior to deploying the techniques in this paper, CALDERA leveraged a hard-coded finite-state machine: by comparison, the planning-based approach is smoother, easier to vary, more adaptable, and much easier to extend, in addition to some efficiency boosts. We note that, because the new approach is much easier to extend and adapt, other researchers were able to use our public implementation of CALDERA to integrate their own techniques – modeling and coding them in our Pythonic representation – with the planner integrating the new actions seamlessly into its operations (Bottomley, P. and Beukema, W. 2018).

Discussion

Our hope with this paper is twofold: first, to call attention to the need for a planning and acting paradigm within the security and planning community, and second, to raise awareness of the particular types of uncertainty considered in the automated adversary emulation problem. With regards to the latter, our primary call-to-action is inspired by our experiences and design goals: we wish to avoid having users explicitly input network parameters, or even network possibilities, when they run their tests. This contrasts with approaches previously identified in the literature, where they assume access to either a network map, or have clearly bounded uncertainty with which they can run traditional planning techniques. In our own modeling efforts, we have found this to be a difficult task.

We note that the approaches discussed in this paper have been designed to exhibit *emergent* behavior, as opposed to explicit goals or alternative execution strategies. While this is similar to real adversaries, who typically have semi-vague goals (e.g., "exfiltrate all sensitive files"), we believe that this is an area for further research. As it stands, the current heuristic approach prioritizes executing "goal actions" as soon as possible. By contrast, some adversaries may prefer to lay-in-wait, achieving their goals on each host simultaneously – i.e., laterally move to all hosts and then encrypt them, as opposed to encrypting them as you move through the network. Similarly, a system that could achieve a specific goal – i.e., compromise a specific host – would be of great utility for defenders. Towards this, the heuristic approach can be modified to exhibit this type of behavior, however it involves a significant amount of manual analysis of both goal and reward, and is not guaranteed to be optimal. Instead, we believe it may be possible to leverage the unique cyber domain properties to construct a planner that can more accurately achieve this. As it stands, our current approach offers a workable solution for smaller problems, but blows up combinatorially as the depth and domain grows.

Most of the formal modeling in the automated planning community of cyber is either domain specific – i.e., network protocols – or heavily focused on exploits. Because adversaries tend to re-use existing functionality, these latter models lack realism; adversaries do not achieve lateral movement only through exploits. Instead, an ideal representation would cover other ordinarily benign activities that adversaries also use. Based on our experiences, this can be challenging to do from the adversary's perspective (i.e., in bounding the adversary's uncertainty). Additionally, while not covered in high detail in this paper, that adversaries create and reason over new domain objects is a similar encoding issue. We are currently developing a translation from our Pythonic model to PDDL(McDermott et al. 1998), but have found that we often need to use unnatural constructs to represent key cyber concepts. We also plan on investigating the use of epistemic planning (Löwe, Pacuit, and Witzel 2011) to represent the adversary's changing belief states – as encoded, maintaining dual states between the world and the adversary's knowledge is cumbersome, and we believe we can leverage existing strategies to optimize our process.

Conclusion Both industry and academia have recognized the utility of automated adversary emulation and penetration testing, the former solving it from an implementation-first perspective, and the latter working on the theory. Neither side, however, seems to have recognized the key challenges that make this a hard problem, nor have others formalized the requirements that an ideal automated offensive solution should meet. We hope that in publishing this paper, we can better characterize these challenges and requirements, helping others better understand the nature of the problem and encouraging future research.

References

[Amos-Binks et al. 2017] Amos-Binks, A.; Clark, J.; Weston, K.; Winters, M.; and Harfoush, K. 2017. Efficient attack plan recognition using automated planning. In *Computers and Communications (ISCC), 2017 IEEE Symposium on*, 1001–1006. IEEE.

[Applebaum et al. 2016] Applebaum, A.; Miller, D.; Strom, B.; Korban, C.; and Wolf, R. 2016. Intelligent, automated red team emulation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, 363–373. ACM.

[Applebaum et al. 2017] Applebaum, A.; Miller, D.; Strom, B.; Foster, H.; and Thomas, C. 2017. Analysis of automated adversary emulation techniques. In *Proceedings of the Summer Simulation Multi-Conference*, 16. Society for Computer Simulation International.

[Bottomley, P. and Beukema, W. 2018] Bottomley, P. and Beukema, W. 2018. Signal the ATT&CK: Part 1. `https://www.pwc.co.uk/issues/cyber-security-data-privacy/research/signal-att-and-ck-part-1.html`.

[Bozic and Wotawa 2017] Bozic, J., and Wotawa, F. 2017. Planning the attack! or how to use ai in security testing? In *IWAISe: First International Workshop on Artificial Intelligence in Security*, 50.

[Durkota 2014] Durkota, K. 2014. Computing optimal policies for attack graphs with action failures and costs. In *STAIRS*, 101–110.

[Eiter et al. 2000] Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2000. *Computational Logic — CL 2000: First International Conference London, UK, July 24–28, 2000 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg. chapter Planning under Incomplete Knowledge, 807–821.

[Ghallab, Nau, and Traverso 2014] Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actors view of automated planning and acting: A position paper. *Artificial Intelligence* 208:1–17.

[Hoffmann 2015] Hoffmann, J. 2015. Simulated penetration testing: From "dijkstra" to "turing test++".

[Katz, Moshkovich, and Karpas 2016] Katz, M.; Moshkovich, D.; and Karpas, E. 2016. Lifting delete relaxation heuristics to successor generator planning. *Heuristics and Search for Domain-independent Planning (HSDIP)* 61.

[Khan and Parkinson 2017] Khan, S., and Parkinson, S. 2017. Towards automated vulnerability assessment.

[Löwe, Pacuit, and Witzel 2011] Löwe, B.; Pacuit, E.; and Witzel, A. 2011. DEL planning and some tractable cases. In *International Workshop on Logic, Rationality and Interaction*, 179–192. Springer.

[McDermott et al. 1998] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

[Obes, Sarraute, and Richarte 2010] Obes, J. L.; Sarraute, C.; and Richarte, G. 2010. Attack planning in the real world. In *Working Notes for the 2010 AAAI Workshop on Intelligent Security (SecArt)*, 10.

[Sarraute, Buffet, and Hoffmann 2012] Sarraute, C.; Buffet, O.; and Hoffmann, J. 2012. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI-12)*.

[Shmaryahu et al. 2017] Shmaryahu, D.; Shani, G.; Hoffmann, J.; and Steinmetz, M. 2017. Partially observable contingent planning for penetration testing. In *IWAISe: First International Workshop on Artificial Intelligence in Security*, 33.

[Smith, Casey 2017] Smith, Casey. 2017. Red Canary Introduces Atomic Red Team, a New Testing Framework for Defenders. `https://www.redcanary.com/blog/atomic-red-team-testing/`.