Reasoning about Sequential Cyberattacks

Vivin Paliath
Arizona State University
Tempe, US
vivin@asu.edu

Paulo Shakarian
Arizona State University
Tempe, US
shak@asu.edu

***Abstract*—Cyber adversaries employ a variety of malware and exploits to attack computer systems, usually via sequential or "chained" attacks, that take advantage of vulnerability dependencies. In this paper, we introduce a formalism to model such attacks. We show that the determination of the set of capabilities gained by an attacker, which also translates to extent to which the system is compromised, corresponds with the convergence of a simple fixed-point operator. We then address the problem of determining the optimal/most-dangerous strategy for a cyber-adversary with respect to this model and find it to be an NP-Complete problem. To address this complexity we utilize an A*-based approach with an admissible heuristic, that incorporates the result of the fixed-point operator and uses memoization for greater efficiency. We provide an implementation and show through a suite of experiments, using both simulated and actual vulnerability data, that this method performs well in practice for identifying adversarial courses of action in this domain. On average, we found that our techniques decrease runtime by 82%.**

***Index Terms*—cybersecurity, cyber-attack modeling, adversarial reasoning**

I. Introduction

Contemporary cyber-threat actors rely on a variety of malware and exploits purchased through various channels such as the darkweb. These exploits are usually part of sophisticated "exploit kits" that target vulnerabilities and leverage their interdependencies to form "chained attacks". In this paper, we introduce a formalism that allows modeling adversarial-actions against a system with possibly-interdependent vulnerabilities using associated exploits. We present an operator that we show has a least fixed-point, which corresponds to the precise set of capabilities obtained by the attacker under our framework and is efficient to compute. We address the problem of determining the optimal/most-dangerous strategy for a cyber-adversary with respect to this model and find it to be NP-Complete. We also present a suite of provably-correct algorithms based on A* search to solve this problem. We develop an admissible heuristic, and show how we enhance algorithm performance by incorporating fixed-point operator results and by using a memoization approach that we also introduce. With these techniques, we see an average performance-improvement of 82% over standard A*. We demonstrate the performance of these algorithms and the validity of our model through a suite of experiments using both simulated and actual vulnerability-data. This paper also includes a brief overview of related work. Additional supporting material, results, and full proofs can be found online[1].

ASU Global Security Initiative (GSI) and the Office of Naval Research (ONR) Neptune program.

ASONAM '19, August 27-30, 2019, Vancouver, Canada

ACM ISBN 978-1-4503-6868-1/10/08…$15.00
https://doi.org/10.1145/3341161.3343522

II. Model

In this section, we define a formal mathematical-model to capture the semantics of chained attacks that leverage vulnerability and exploit interdependencies. The model assumes that there is an attacker with an initial set of "capabilities". These capabilities represent system functionality that the attacker has access to, and also represent different actions that the attacker can take on or against the system. Using different sets of initial capabilities, we can model various attacker-types, such as unauthenticated/authenticated remote-attackers, local users, etc. The set of initial capabilities can also reflect prior knowledge gained by scanning or profiling a target system. Using their capabilities the attacker employs a sequence of exploits, where each subsequent exploit uses capabilities provided by a previous one. Through this iterative approach, the attacker gradually compromises the system until they have achieved their objective.

We first define a set V, which is the set of all vulnerabilities, and a set C, which is the set of capabilities supported by a component on the information systems network. We use the notation 2^C to represent the powerset of C. Following the intuition of [1]–[5], we assume each exploit has two sets of capabilities: C^r representing preconditions and C^g representing postconditions. These capabilities can represent high-level interactions (e.g., HTTP requests), local commands on the system, and malicious capabilities gained by successfully using an exploit:

Definition 1.1. Given a set $C^r \subseteq C$ of capabilities required to use an exploit for a vulnerability $v \in V$, and a set $C^g \subseteq C$ of capabilities gained by using that exploit, the set of exploits E is a set of tuples of the form (C^r, C^g, v).

[1] http://vivin.net/pub/neptune-supplemental.pdf

This formalism captures the necessary conditions required to use a particular vulnerability exploit, and also models what the attacker gains by using the exploit. It also lets us model the attack surface of the system. For example, we can model an "exploit" that grants privileged capabilities with the precondition that the user is able to escalate their privilege. Similarly, a "passive" attack-surface can also be modeled as an "exploit" with no required capabilities; this is useful to model services running on a system, such as web servers. We acknowledge that deriving these capability sets from CVE descriptions is challenging. However, there are approaches [6], [7] that have solved this problem with some success, which can be easily adapted to work with our model. Note that it is possible for multiple exploits to target a single vulnerability.

We now define an exploit chain as follows:

Definition 1.2. Given some set $E' \subseteq E$, and initial set of capabilities $C_0 \subseteq C$, an **exploit chain** (denoted $E_{C_0,E'}$) is a subset of E':

$\langle e_n \mid n \in \mathbb{N} \rangle = \langle (C_1^r, C_1^g, v_1), \ldots, (C_i^r, C_i^g, v_i), \ldots, (C_n^r, C_n^g, v_n) \rangle$
where:
$C_1^r \subseteq C_0$
and for each consecutive pair $(C_i^r, C_i^g, v_i), (C_j^r, C_j^g, v_j)$ we have:
$C_j^r \subseteq C_0 \cup \bigcup_{k<j} C_k^g$

An attacker can use different exploit chains to achieve various goals; there may even be multiple exploit-chains that lead to the same goal. By accounting for all these chains, we can identify the complete set of capabilities that the attacker can gain; we call this the **obtained set of capabilities**. The intuition is that this set establishes an upper bound on the set of capabilities an attacker can gain, for a given set of initial capabilities and exploits. It easily translates to the maximum amount of damage an attacker can cause to the system.

Definition 1.3. Given a set of exploits $E' \subseteq E$ and a set of initial capabilities $C_0 \subseteq C$, the **obtained set of capabilities** (denoted C^*_{E',C_0}) consists of all capabilities in C_0, in addition to any capability $c \in C$ for which there exists an exploit chain $E_{C_0,E'}$ leading to c, and does not contain any capability $c' \in C$ such that $c' \notin C_0$, and for which there exists no exploit chain $E_{C_0,E'}$ leading to c'.

We now design an operator that models exploit application and prove that it has a fixed point, that corresponds to the **obtained set of capabilities**. We also show that the operator converges in a polynomial number of steps and investigate some other useful properties. A key assumption that we make here is that exploit application is monotonic; applying an exploit does not remove a previously-obtained capability. This assumption is quite common in related work [2], [8], [9] in this area. In [10], the authors also note that non-monotonic attacks can be treated as monotonic, if certain low-level details are ignored. As such information is not provided, or difficult to obtain from CVE and exploit descriptions, the monotonicity assumption is reasonable.

Definition 1.4. We introduce the notion of exploit application via an "exploit-application operator":

i. Given $E' \subseteq E$ and $C_0 \in 2^C$ we define the exploit-application operator $\mathbf{T}_{E'} : 2^C \to 2^C$ as:

$$\mathbf{T}_{E'}(C_0) = C_0 \cup \bigcup \{C^g \mid (C^r, C^g, v) \in E' \wedge C^r \subseteq C_0\}$$

ii. Given some $i \in \mathbb{N}$, we define the i^{th} application of $\mathbf{T}_{E'}$ on $C_0 \subseteq C$ as:

$$\mathbf{T}_{E'}{\uparrow_i}(C_0) = \mathbf{T}_{E'}(\mathbf{T}_{E'}{\uparrow_{i-1}}(C_0))$$
where:
$$\mathbf{T}_{E'}{\uparrow_1}(C_0) = \mathbf{T}_{E'}(C_0)$$

iii. $\mathbf{T}^*_{E'}$, the fixed point of $\mathbf{T}_{E'}$ is defined as:

$$\mathbf{T}^*_{E'}(C_0) = \mathbf{T}_{E'}{\uparrow_i}(C_0)$$
where:
$$\mathbf{T}_{E'}{\uparrow_i}(C_0) = \mathbf{T}_{E'}{\uparrow_{i+1}}(C_0)$$

In the above definition, item (i) describes a single application of the operator, which augments the attacker's initial capabilities using C^g from only those exploits where $C^r \subseteq C_0$. Item (ii) models the iterative behavior of the attacker; on each subsequent application of the operator, it uses the augmented capability-set from the previous application, potentially allowing the attacker to exploit additional vulnerabilities. Item (iii) defines the end of this iterative process by defining the fixed-point of the operator, implying that the iterative process must converge, or in real-world terms, the attacker stops when they cannot gain any more capabilities. We can easily prove that this fixed-point exists using lattice theory by first making the following observation about 2^C:

Observation 1.1. $\langle \subseteq, 2^C \rangle$ is a partial ordering and 2^C specifies a complete lattice.

This is straightforward as $\subseteq$ is clearly reflexive, transitive, and antisymmetric. Further, the set 2^C has a clear top element (set C) and a bottom element (the empty set). As C is simply a set of elements, the powerset as under the ordering relationship specified by $\subseteq$ is the classic example of a complete lattice.

Theorem 1.1. *$\mathbf{T}_{E'}$ has a least fixed point.*

The set of capabilities at the least fixed-point represents the complete set of capabilities that the attacker can gain for a given E' and C_0. This is useful because this set corresponds to the **obtained set of capabilities** (Definition 1.3):

Theorem 1.2. *Given an $E' \subseteq E$ and $C_0 \in 2^C$ with obtained set of capabilities C^*_{E',C_0}, $\mathbf{T}^*_{E'}(C_0) \subseteq C^*_{E',C_0}$ and $\mathbf{T}^*_{E'}(C_0) \supseteq C^*_{E',C_0}$.*

A useful property of the operator is that it also identifies mutually disjoint exploit-sets associated with each application:

Definition 1.5. Given a $E' \subseteq E$ and $C_0 \in 2^C$ with an $i \in \mathbb{N}$ such that $\mathbf{T}^*_{E'}(C_0) = \mathbf{T}_{E'}{\uparrow_i}(C_0)$, we can identify the set of

exploits used at any $j \in \mathbb{N}$ where $j \leq i$ as:

$$E'_j = \{(C^r, C^g, v) \in E' \setminus \bigcup_{k=1}^{j-1} E'_k \mid C^r \subseteq \mathbf{T}_{E'} \uparrow_{j-1}(C_0)\}$$

where:

$$E'_1 = \{(C^r, C^g, v) \in E' \mid C^r \subseteq C_0\}$$

We can use this to show that the operator converges in a polynomial number of steps:

Proposition 1.1. *Given $E' \subseteq E$ and $C_0 \in 2^C$ with $i \in \mathbb{N}$ such that $\mathbf{T}^*_{E'}(C_0) = \mathbf{T}_{E'} \uparrow_i (C_0)$, $i \leq |E'|$.*

Exploit sets identified in Definition 1.5 have some interesting properties. Observe that the operator effectively "sorts" E' into an ordered sequence of subsets using dependency information; an exploit in E'_j will have required capabilities provided by at least one exploit from E'_{j-1}. This gives us information about the structure of possible exploit-chains as we now have a general idea about the order in which we expect attackers to use certain exploits.

We now investigate the complexity of calculating the fixed point. Observe that for a given C_0 and E', the complexity of calculating $\mathbf{T}_{E'}(C_0)$ is $\Theta(|E'|)$ as we iterate over the entire set. Given a set $E'' \supseteq E'$, observe that we can calculate $\mathbf{T}_{E''}(C_0)$ using $\mathbf{T}_{E'}(C_0)$ via the expression $\mathbf{T}_{E'}(C_0) \cup \mathbf{T}_{E'' \setminus E'}(C_0)$. Hence, we can state the following proposition:

Proposition 1.2. *Given $E'' \subseteq E$, $E' \subseteq E$ and $C_0 \in 2^C$, if $E' \subseteq E''$ then $\mathbf{T}_{E'}(C_0) \subseteq \mathbf{T}_{E''}(C_0)$*

We also extend this memoization to the calculation of the fixed-point. Note that the complexity here is $\Theta(k\,|E'|)$, where k is the number of applications taken to reach the fixed-point ($1 \leq k \leq |E'|$). As before, given a set $E'' \supseteq E'$, we can intuitively see that it is possible to use the result of $\mathbf{T}^*_{E'}(C_0)$ when calculating $\mathbf{T}^*_{E''}(C_0)$. Hence:

Proposition 1.3. *Given $E'' \subseteq E$, $E' \subseteq E$ and $C_0 \in 2^C$, if $E' \subseteq E''$ then $\mathbf{T}^*_{E'}(C_0) \subseteq \mathbf{T}^*_{E''}(C_0)$*

We will now try to express $\mathbf{T}^*_{E''}(C_0)$ in terms of $\mathbf{T}^*_{E'}(C_0)$. Since we do not necessarily use every exploit in E' when calculating $\mathbf{T}^*_{E'}(C_0)$, we cannot exclude E' outright as with the memoization of $\mathbf{T}_{E''}(C_0)$. This is because exploits in $E'' \setminus E'$ may provide capabilities that can make previously unused exploits from E' applicable. Hence we must exclude only those exploits that were used to calculate $\mathbf{T}^*_{E'}(C_0)$. We define these exploits as follows:

Definition 1.6. Given a $E' \subseteq E$ and $C_0 \in 2^C$, with an $i \in \mathbb{N}$ such that $\mathbf{T}^*_{E'}(C_0) = \mathbf{T}_{E'} \uparrow_i (C_0)$, the complete set of applicable exploits E_{use} is defined as:

$$E_{use} = \bigcup_{1 \leq j \leq i} E'_j \qquad \text{(Definition 1.5)}$$

We can now express $\mathbf{T}^*_{E''}(C_0)$ in terms of $\mathbf{T}^*_{E'}(C_0)$ as $\mathbf{T}^*_{E''}(C_0) = \mathbf{T}^*_{E'' \setminus E_{use}}(\mathbf{T}^*_{E'}(C_0))$. Notice that the runtime is now $\Theta(k'\,|E'' \setminus E_{use}|)$ instead of $\Theta(k\,|E''|)$ (where $k' \leq k$).

We now address the problem of identifying the attacker's strategy. Attackers decide what exploits to use by profiling a system, or by having knowledge about the kinds of systems used in a network. With this information, they can use public vulnerability databases to look up associated vulnerabilities providing specific capabilities. However, the attacker may not have the expertise necessary to exploit such vulnerabilities and could consider purchasing exploit kits from the darkweb. Note that by using the darkweb as a resource, it is possible for the attacker to purchase exploit-kits that target undisclosed (zero-day) vulnerabilities as well. Both expertise and financial cost can be expressed using a cost function:

Definition 1.7. Given a set of exploits E, we define a cost function $cost : E \to \mathbb{R}^+$ that associates a real-valued cost with each exploit.

For simplicity, we use a single cost-function throughout this paper, but all of the results can be extended for separate ones. Cost functions can also take into account other factors, such as the probability that a vulnerability has been patched, as attackers may not have perfect knowledge about a system. This information can be used by the attacker to "weight" the costs of associated exploits in order to factor in this probability. In ongoing work, we are currently looking at data-driven cost-functions [11], [12] as well.

Since attackers do not have unlimited resources, it makes sense to define a budget. Attackers would prefer to use a set of exploits that let them achieve their goal without exceeding this budget. This is the attacker's preferred strategy and is defined as follows:

Definition 1.8. Given the attacker's budget $b \in \mathbb{R}^+$, set of desired capabilities C_d, and initial set of capabilities C_0, the **preferred attack-strategy** $\text{PAS}(C_0, C_d)$ is the set of exploits E' satisfying the following conditions:

- Coverage: $\mathbf{T}^*_{E'}(C_0) \supseteq C_d$
- Cost: $\sum_{e \in E'} cost(e) \leq b$

In the optimization variant of the preferred attack-strategy problem, the quantity $\sum_{e \in E'} cost(e)$ is minimized, and the associated strategy is called an ***optimal strategy***.

Finding a preferred attack-strategy is NP-complete, and is easily shown by a reduction from set cover.

Theorem 1.3. *Finding a preferred attack-strategy is NP-complete.*

III. Algorithms

In this section we examine algorithms to solve the preferred attack-strategy problem. Our baseline approach used depth-first search across the strategy-space. A major shortcoming with this approach is that it has exponential-time complexity and an optimal solution is not guaranteed. It is also obvious that the algorithm explores paths that are unproductive or invalid; for example, paths containing exploits whose preconditions have not been met, or exploits that provide capabilities

the attacker has already obtained. To address these issues we enable more-efficient searches for attacker strategies through the following improvements that maintain correctness:

1. We correctly prune the available exploits at each step and employ the use of an admissible heuristic-function by adopting A* search.
2. We further improve A* search by using the results of the fixed-point operator.

A. Pruning Exploits

The depth-first search approach considers exploits that need not or cannot be part of the solution. To address this problem we prune the search-tree by discarding such exploits. An exploit is included and not pruned only if the following conditions hold:

1. The exploit does not cause the attacker to exceed their budget.
2. The attacker currently has the required set of capabilities to apply the exploit.
3. The exploit offers at least one capability that the attacker does not already have.

The correctness of this pruning technique follows directly from our original model.

B. PAS-A*

While pruning helps us address time-complexity and can provide better solutions than DFS, it still does not guarantee an optimal solution. To address this issue we use A* search (Algorithm 1). With an admissible heuristic, the tree-search variant of A* is both complete and optimal. A* evaluates nodes by combining $g(e)$, the cost to reach the node, and $h(e)$, the lower bound of the cost to get from the node to the goal. To estimate the cost to the goal, we first look at how many desired capabilities remain to be gained:

Definition 1.9. Given the attacker's current set of exploits E', initial set of capabilities C_0, desired set of capabilities C_d, and the applicable exploit under consideration (C^r, C^g, v), we define the remaining capabilities as $C_{rem} = C_d \setminus \mathbf{T}^*_{E'}(C_0) \setminus C^g$.

These capabilities must come from the set of remaining applicable exploits, which we define as follows:

Definition 1.10. Given the complete set of exploits E, the attacker's current set of exploits E', and the applicable exploit under consideration e, we define the remaining set of exploits as $E_{rem} = \{(C^r, C^g, v) \in E \setminus E' \setminus \{e\} \mid C^g \cap C_{rem} \neq \varnothing\}$.

Note that if $E_{rem} = \varnothing$ while $C_{rem} \neq \varnothing$, it implies that there is no solution since there are no remaining exploits that provide any of the remaining desired-capabilities.

We now estimate the cost to the goal by identifying the lowest-possible cost to gain each remaining capability, and then summing those costs:

Definition 1.11. Given the node e that represents an applicable exploit under consideration, the remaining set of capabilities C_{rem}, and the remaining set of exploits E_{rem}, the estimated cost to the goal $h : E \to \mathbb{R}^+$ is defined as:

$$h(e) = \sum_{c \in C_{rem}} \min_{\{e' \in E_{rem} | c \in C^g\}} \frac{cost(e)}{|C^g \cap C_{rem}|}$$
$$\text{where } e' = (C^r, C^g, v)$$

Theorem 1.4. *$h(e)$ is admissible.*

Algorithm 2 is an implementation of $h(e)$.

Algorithm 1 Implementation of PAS-A*

```
1:  procedure PAS-A*(E, C_0, C_d, b):
2:      function PATH(node):
3:          path as SET
4:          while node.PARENT ≠ ∅ do:
5:              ADD(path, node.EXPLOIT)
6:              node ← node.PARENT
7:          return path
8:      function SOLUTION(node):
9:          return PATH(node)
10:     function ROOT:
11:         root as NODE
12:         root.PARENT ← ∅
13:         root.EXPLOIT ← ∅
14:         root.PATHCOST ← 0
15:         return root
16:     function MAKENODE(parent, e):
17:         node as NODE
18:         node.PARENT ← parent
19:         node.EXPLOIT ← e
20:         node.PATHCOST ← parent.PATHCOST + cost(e)
21:         return node
22:     function PRUNE(parent, e):
23:         if parent.PATHCOST + cost(e) > b then:
24:             return true
25:         else if e.C^r ⊈ T*_E'(C_0) then:
26:             return true
27:         else if e.C^g ⊆ T*_E'(C_0) then:
28:             return true
29:         return false
30:     function EXPAND(node):
31:         children as SET
32:         for each e in E \ PATH(node):
33:             if not PRUNE(node, e) then:
34:                 ADD(children, MAKENODE(node, e))
35:         return children
36:     function REMAININGCAPABILITIES(node):
37:         E' ← PATH(node.PARENT)
38:         return C_d \ T*_E'(C_0) \ C^g
39:     function REMAININGEXPLOITS(node, C_rem):
40:         E_rem as SET
41:         for each e in E \ PATH(node):
42:             if e.C^g ∩ C_rem ≠ ∅ then:
43:                 ADD(E_rem, e)
44:         return E_rem
45:     function ESTIMATEDCOST(node):
46:         C_rem ← REMAININGCAPABILITIES(node)
47:         E_rem ← REMAININGEXPLOITS(node, C_rem)
48:         return node.PATHCOST + h(C_rem, E_rem)
49:     nodes as PRIORITY-QUEUE ordered by ESTIMATEDCOST
50:     ENQUEUE(nodes, ROOT)
51:     loop do:
52:         if EMPTY(nodes) then:
53:             return failure
54:         node ← DEQUEUE(nodes)
55:         E' ← SOLUTION(node)
56:         if T*_E'(C_0) ⊇ C_d then:
57:             return E'
58:         for each child in EXPAND(node):
59:             if not EXISTS(nodes, child) then:
60:                 ENQUEUE(nodes, child)
61:             else:
62:                 existing ← FIND(nodes, child)
63:                 f_existing ← ESTIMATEDCOST(existing)
64:                 f_child ← ESTIMATEDCOST(child)
65:                 if f_child < f_existing then:
66:                     REPLACE(nodes, existing, child)
```

Algorithm 2 Implementation of heuristic h

```
1:  function h(C_rem, E_rem):
2:      h as REAL
3:      h ← 0
4:      for each c in C_rem:
5:          h_min ← ∞
6:          for each e in E_rem:
7:              if c ∈ e.C^g then:
8:                  h_e ← cost(e) ÷ |e.C^g ∩ C_rem|
9:                  if h_e < h_min then:
10:                     h_min ← h_e
11:         h ← h + h_min
12:     return h
```

Algorithm 3 Enhanced A* (common functions omitted)

```
1:  procedure PAS-ENHANCED-A*(E, C_0, C_d, b):
2:      function PRUNE(parent, e):
3:          if e.i < parent.EXPLOIT.i or e.i − parent.EXPLOIT.i > 1 then:
4:              return true
5:          else if parent.PATHCOST + cost(e) > b then:
6:              return true
7:          else if e.C^r ⊈ T*_{E'}(C_0) then:
8:              return true
9:          else if e.C^g ⊆ T*_{E'}(C_0) then:
10:             return true
11:         return false
12:     function REMAININGEXPLOITS(node, C_rem):
13:         E_rem as SET
14:         for each e in E \ PATH(node):
15:             if e.i ≥ node.EXPLOIT.i and e.C^g ∩ C_rem ≠ ∅ then:
16:                 ADD(E_rem, e)
17:         return E_rem
18:     ⟨E_use, C⟩ ← T*_E(C_0)        ▷ Assume that fixed-point operator returns used exploits.
                                       Implementation is trivial and has been left out for brevity.
19:     E ← E_use                     ▷ Redefine E to include only used exploits.
20:     ▷ Remaining code identical to PAS-A*
```

C. Enhancing PAS-A* using fixed-point operator results

While PAS-A* identifies optimal solutions, we can improve its runtime performance by making use of the fixed-point operator's results. Observe that Definition 1.6 lets us identify the complete set of exploits that an attacker can possibly use, given their initial set of capabilities. Hence the preferred strategy can never contain an exploit that exists in $E' \setminus E_{use}$. By having PAS-A* consider only those exploits in E_{use}, we can reduce the search space. Additional improvements can be gained using Definition 1.5, which identifies the set of exploits used at each application. By labeling each exploit from E_{use} with the application in which it was used, we can prune the search tree even further. Assume that we are expanding a node associated with exploit e^i, which was used in the i^{th} application. We can then prune any exploit e^j where $j < i$ or $j > i + 1$, because a subsequent exploit must have either been used in the same application as the preceding one, or in the very next application. We also use this information to improve heuristic performance by excluding all remaining exploits that were used in applications earlier than the exploit under consideration. Algorithm 3 is an implementation of PAS-ENHANCED-A*.

D. Improving fixed-point operator performance

We implement fixed-point operator memoization by means of a trie-based [13] cache. We first associate a unique integer with each exploit, allowing us to order a set of them consistently. We can then cache the results in a trie (see Figure 1 for an example), giving us lookup time that is linear in the size of the largest subset of E' cached so far. Note that in this approach, we assume that C_0 is fixed. While the upper bound on the operator's complexity given some arbitrary E' is quadratic even with caching, it is effectively linear when used with algorithms PAS-A* and PAS-ENHANCED-A*. As the path to any node in the search tree only consists of applicable exploits, every exploit is used when calculating the fixed-point ($E_{use} = E'$). Furthermore we grow the path a single exploit at a time, making the complexity of calculating $\mathbf{T}^*_{E''}(C_0)$ (where $E'' = E' \cup \{e\}$) just $O(|E'| + 1)$ or $O(|E''|)$.

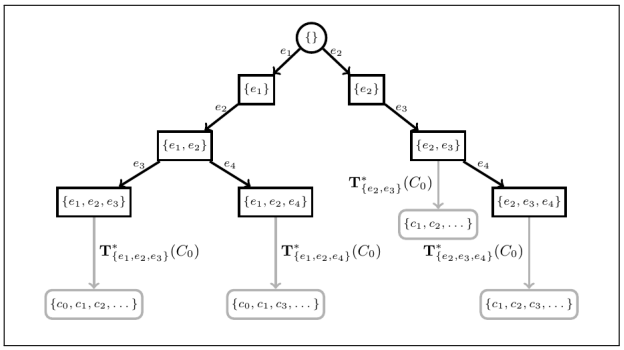

Fig. 1: Example of a trie used to cache results of the fixed-point operator.

IV. Experiments and Results

Our experiments were designed to evaluate performance and to examine the viability of our model and its algorithms in real-world scenarios. All experiments except one were implemented in Java. The remaining experiment, which analyzed the performance of our memoization approach, was implemented in JavaScript on NodeJS. All experiments were executed on a machine with an Intel Core i7-4770K processor and 24GB RAM running Linux Mint 18.3.

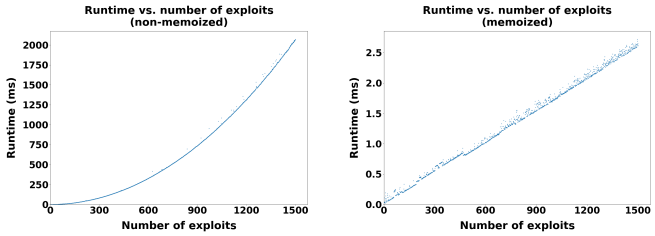

Fig. 2: Runtime vs. number of exploits for non-memoized and memoized $\mathbf{T}^*_{E'}$.

Our first experiment assesses the impact of memoization on $\mathbf{T}^*_{E'}$. We generated a set of exploits $E' = \{e_0, e_1, \dots, e_n\}$ of the form $\{(\varnothing, \{c_0\}, v_0), (\{c_0\}, \{c_1\}, v_1), \dots, (\{c_{n-1}\}, \{c_n\}, v_n)\}$. We then measured the runtime of memoized and non-memoized versions of $\mathbf{T}^*_{E'}$ against the following sequence of subsets of E': $\langle \{e_0\}, \{e_0, e_1\}, \dots, \{e_0, e_1, \dots, e_i\}, \dots, \{e_0, e_1, \dots, e_n\} \rangle$. From the results in Figure 2, it is evident that runtime for the non-memoized version is quadratic, compared to linear for the memoized version. This confirms that memoization significantly improves the performance of $\mathbf{T}^*_{E'}$.

The next set of experiments use real-world vulnerability data gathered from the National Institute of Standards and Technology's National Vulnerability Database (NIST NVD). We defined 22 exploits based on 10 Critical Vulnerability Exploits (CVEs) associated with the Windows 10 operating system. We chose CVEs that involve a diverse spectrum of malicious capabilities and also feature a good distribution of attacker types and complexity levels. The cost of each exploit was set to the sum of the attack and access complexities

of the associated CVE; these metrics are extracted from the Common Vulnerability Scoring System (CVSS) base-score vector reported for each CVE in the NVD. Both types of complexities are graded as High, Medium, or Low, which we associated with integer values 3, 2, and 1. We chose these specific metrics as they describe how easy or difficult it is to exploit a particular vulnerability. Exploit capability-sets were defined manually, based on CVE descriptions and CVSS vector attributes. We also defined four types of attackers: a **remote attacker** with remote access to the machine, an **authenticated remote-attacker** with remote access and valid credentials for the machine, a **physically-proximate attacker** with physical and network access to the machine, and a **local attacker** with physical access and valid credentials for local use of the machine. C_d for all attackers was defined to contain the following capabilities: **remote code-execution**, **privilege escalation**, **installing a crafted boot-manager**, and **denial of service**. C_0 for each attacker was defined to reflect their level of access to the machine, with the remote attacker having the least number of initial capabilities and the local attacker having the most.

Attacker Type	$\lvert\mathbf{T}^*_E(C_0)\rvert$	$\lvert E'\rvert$	Cost
Remote	13	10	20
Remote authenticated	13	8	16
Physically proximate	18	9	15
Local	22	7	11

TABLE I: Number of obtained capabilities, and size and cost of optimal solution based on attacker type.

Our hypothesis for this experiment was that attackers with greater initial-access to the system would be able to cause more damage than those with lower access, given identical sets of exploits. The results (Table I) clearly validate this hypothesis, as attackers with a greater number, or with more significant initial-capabilities, obtain a larger set of capabilities. This also shows that the results of the fixed-point operator are realistic and match real-world expectations.

The next experiment investigates how the preferred strategy (as identified by PAS-A*) varies across attacker types. Each attacker's budget was set to be unlimited. The results (Table I) show that in general, solution costs are inversely proportional to the number and significance of initial capabilities. This is to be expected, as attackers with more initial access to a system would be able to compromise it using fewer resources. These results provide additional confirmation that the model is able to produce results that are not only accurate, but also make sense intuitively and reflect real-world attacker behavior.

The aim of our next experiment was to examine the effectiveness of our pruning approach by running PAS-DFS and PAS-DFS-PRUNED against each attacker type, initially with unlimited budgets. The results (Figure 3) show that while the average runtime of PAS-DFS-PRUNED is generally not lower than PAS-DFS, the solutions identified are cheaper and qualitatively better. The low runtimes of PAS-DFS are easily explained by the fact that with an unlimited budget, a

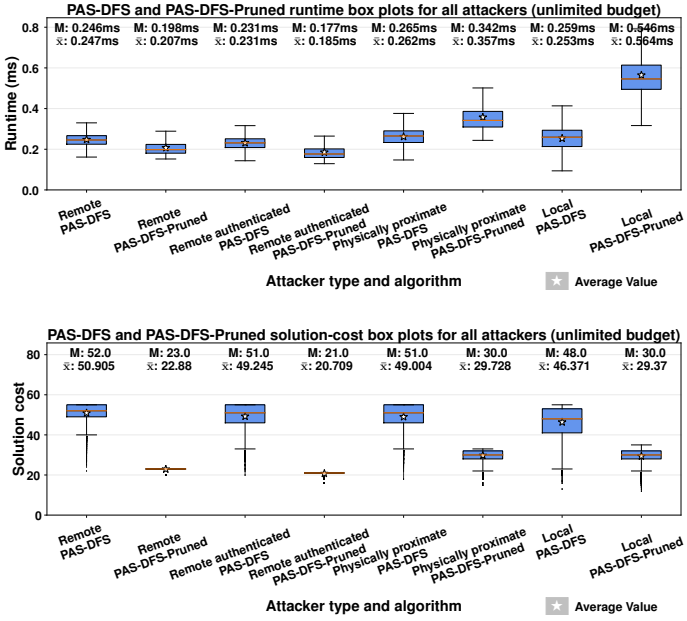

Fig. 3: Box plot of runtimes and solution costs for PAS-DFS and PAS-DFS-PRUNED, against different attacker types (1000 samples each) with unlimited budgets. Some outliers excluded for clarity.

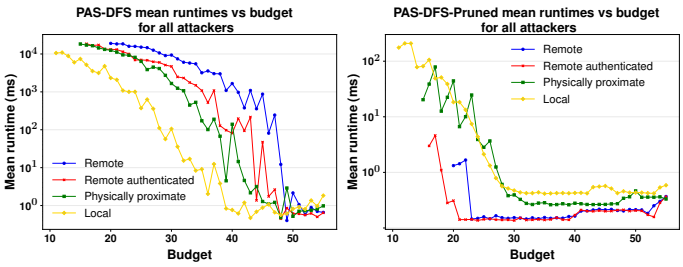

Fig. 4: Mean runtime vs budget for PAS-DFS and PAS-DFS-PRUNED against different attacker types with restricted budgets (100 and 1000 samples respectively).

degenerate solution (one that contains all exploits) is likely to be quickly identified. In particular, note that the average cost of solutions identified by PAS-DFS is quite close to the total exploit-cost of 55, showing that a large proportion of solutions identified include most or all exploits. This is also evidence of the effectiveness of pruning, as PAS-DFS-PRUNED discards paths that contain such solutions which, though they provide the attacker with their desired capabilities, are typically more expensive as they include redundant or invalid exploits (see items 2 and 3 in III-A). The runtime advantage of pruning is more evident if budgets are restricted to be lower than the total cost of all available exploits. We looked at the mean runtime as a function of the budget for each algorithm, against each attacker type, by varying the budget from 55 (the total cost of all exploits) down to the cost of the optimal solution for each attacker. We observed that in many cases, especially as the budgets approached the cost of the optimal

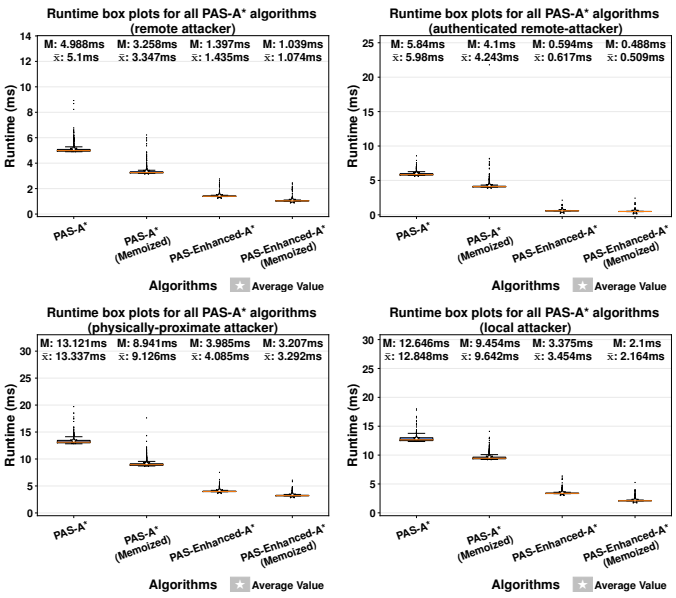

Fig. 5: Box plot of runtimes for all PAS-A* algorithms (memoized and non-memoized) against different attacker types (1000 samples each).

solution, PAS-DFS took more than an hour to complete. Hence we capped the execution time to 20 seconds, which is an order of magnitude higher than the highest-observed runtime for PAS-DFS-PRUNED (2 seconds). The results (Figure 4) clearly show that PAS-DFS-PRUNED is much faster on average. With restricted budgets that are lower than the total cost of all available exploits, PAS-DFS cannot identify degenerate solutions and must therefore explore more, usually unproductive paths, that PAS-DFS-PRUNED ignores. As the budget approaches the cost of the optimal solution, the runtime advantage of PAS-DFS-PRUNED is especially evident. These results show that our pruning approach significantly lowers the size of the search space, and helps identify solutions that are much cheaper and of higher quality than those identified by PAS-DFS.

In our next experiment, we investigate how memoization and attacker type affect performance. Attacker budgets were once again set to be unlimited. As expected, the results (Figure 5) show that memoization improves runtime in all cases; on average we see a 27% decrease in runtime over non-memoized implementations. Generally, runtimes also appear to be directly proportional to the number of initial capabilities. This is because more initial capabilities allow more exploits to be applicable, resulting in a larger search-space due to an increased branching-factor. Finally, we see that the memoized implementation of PAS-ENHANCED-A* performs the best, with a runtime that is on average 82% lower than standard A* (PAS-A* without memoization). When comparing to the baseline approach (Figures 3 and 4), we can see that with unrestricted budgets, runtimes for all PAS-A* variants are higher; this is to be expected as there are far more sub-optimal solutions than optimal ones, and thus it is more likely that one would be found quickly by PAS-DFS or PAS-DFS-PRUNED. However, we can clearly see that when budgets approach the cost of the optimal solution, PAS-DFS is in many cases unable to find a solution in a reasonable time, whereas PAS-DFS-PRUNED is able to find a solution, but takes far longer than any of the PAS-A* algorithms. These results clearly show that PAS-A* and its variants are far superior to the baseline approach.

The final experiment investigates how the attacker's skill-level affects the solution. The initial budget for all attackers is set to the size of their optimal solution as determined from previous experiments. To model attackers with different skill-levels, we multiply their initial budget by an associated factor: 1 for low skill, 2 for medium skill, and 3 for highly skilled. The results (Table II) suggest that attackers need to have a medium skill-level at the very least, in order to compromise the system. This also makes sense intuitively, as we expect attackers with greater expertise to be able to exploit more-complex vulnerabilities.

Attacker Type	Low skill	Medium skill	High skill
Remote	None	20 ($\|E'\| = 10$)	20 ($\|E'\| = 10$)
Remote authenticated	None	16 ($\|E'\| = 8$)	16 ($\|E'\| = 8$)
Physically proximate	None	15 ($\|E'\| = 9$)	15 ($\|E'\| = 9$)
Local	None	11 ($\|E'\| = 7$)	11 ($\|E'\| = 7$)

TABLE II: Solution costs and sizes for attackers of different skill-levels.

V. RELATED WORK

There has been previous work that studied the problem of sequential attack-sequences using exploit preconditions and postconditions. However, the approach in this paper offers several advantages. First, our approach differs from [1], [4] in that it returns a constructive result by providing actual attack-sequences. Second, we have shown that queries can be answered efficiently, which is a favorable advantage over more computationally-intensive approaches [1]–[5]. Third, our approach does not require generating explicit attack-graphs or the enumeration of all-possible attack scenarios, which again makes it far less computationally-intensive compared to approaches that do [2]–[5]. We also note that while there are approaches [8], [14] that use scalable attack-graph generation techniques [10], our approach still offers advantages. In [14] impact-assessments are done on a per-scenario basis, and a comprehensive assessment would require enumerating over all-possible attack scenarios. Whereas in [8], identifying possible attack-sequences requires a breadth-first traversal of the attack-graph, which in the case of large, or poorly-secured networks, can be memory intensive; in comparison, we use A^* search with an efficient heuristic, and also leverage fixed-point operator results to prune the search tree, making our approach less resource-intensive. Finally, we note that in contrast to domain-specific approaches that focus on specific situations, such as attacks on smart grids [1] or web applications [5], our approach is general and can be applied to multiple situations, including networked systems.

VI. Conclusion

In this paper, we developed a framework for modeling sequential-cyberattacks based on vulnerability dependencies. We showed that the attacker's obtained set of capabilities corresponds precisely with the result of a fixed-point computation, and that an actual attack-strategy can be computed efficiently via an A*-based approach, using heuristics that we have developed. Results presented in this paper demonstrate that our model and its algorithms are viable in practice, and that solutions can be easily interpreted according to an intuitive, real-world understanding of attacker behavior. In the future, we intend to augment the selection of vulnerabilities by considering external sources of information, such as the darkweb, and investigate methods to automatically generate capability-sets from CVE descriptions and other sources of information, such as Metasploit modules.

References

[1] T. M. Chen, J. C. Sanchez-Aarnoutse, and J. Buford, "Petri net modeling of cyber-physical attacks on smart grid," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 741–749, 2011.

[2] W. Li, R. B. Vaughn, and Y. S. Dandass, "An approach to model network exploitations using exploitation graphs," *Simulation*, vol. 82, no. 8, pp. 523–541, 2006.

[3] J. Dawkins and J. Hale, "A systematic approach to multi-stage network attack analysis," in *Information Assurance Workshop, 2004. Proceedings. Second IEEE International*. IEEE, 2004, pp. 48–56.

[4] O. Sheyner and J. Wing, "Tools for generating and analyzing attack graphs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2003, pp. 344–371.

[5] E. Alata, M. Kaâniche, V. Nicomette, and R. Akrout, "An automated approach to generate web applications attack scenarios," in *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*. IEEE, 2013, pp. 78–85.

[6] R. Schuppenies, C. Meinel, and F. Cheng, "Automatic extraction of vulnerability information for attack graphs," *Hasso-Plattner-Institute for IT Systems Engineering, University of Potsdam*, 2009.

[7] S. Roschke, F. Cheng, R. Schuppenies, and C. Meinel, "Towards unifying vulnerability information for attack graph construction," in *International Conference on Information Security*. Springer, 2009, pp. 218–233.

[8] S. Jajodia, S. Noel, and B. O'Berry, "Topological analysis of network attack vulnerability," in *Managing Cyber Threats*. Springer, 2005, pp. 247–266.

[9] L. Wang, S. Noel, and S. Jajodia, "Minimum-cost network hardening using attack graphs," *Computer Communications*, vol. 29, no. 18, pp. 3812–3824, 2006.

[10] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 336–345.

[11] J. J. Robertson, V. Paliath, J. Shakarian, A. Thart, and P. Shakarian, "Data driven game theoretic cyber threat mitigation," *Innovative Applications of Artificial Intelligence*, vol. 28, 2016.

[12] J. Robertson, A. Diab, E. Marin, E. Nunes, V. Paliath, J. Shakarian, and P. Shakarian, "Darknet mining and game theory for enhanced cyber threat intelligence," *Cyber Defense Review*, vol. 1, no. 2, 2016.

[13] R. De La Briandais, "File searching using variable length keys," in *Papers presented at the the March 3-5, 1959, western joint computer conference*. ACM, 1959, pp. 295–298.

[14] M. Albanese and S. Jajodia, "A graphical model to assess the impact of multi-step attacks," *The Journal of Defense Modeling and Simulation*, vol. 15, no. 1, pp. 79–93, 2018.