Insider Threat Simulation Through Ant Colonies and ProB

Akram Idani[0000−0003−2267−3639], Aurélien Pepin, and Mariem Triki

Univ. Grenoble Alpes, Grenoble INP, CNRS, F-38000 Grenoble France
akram.idani@univ-grenoble-alpes.fr, aurelien.pepin@grenoble-inp.org, mariem.triki@grenoble-inp.org

Abstract. In cyber-security, insider threats are particularly challenging to prevent because they are carried out by individuals who already have legitimate access to the information system (IS). In fact, insiders exploit their privileges to perform malicious actions. In previous works we proposed to tackle this problem via a backward symbolic search built on a formal B specification of the IS. Unfortunately this approach is not performant because many proof obligations and constraints must be solved interactively. In this paper, we provide a heuristic-based forward search built on the ant colony optimization algorithm called *API* (Ant-based Path Identification) that we implemented using ProB. We show how API can be used to search for malicious scenarios and we present the results of our experiments in comparison with other approaches.

Keywords: B Method · Access Control · Ant colonies · Insider attacks

1 Introduction

Cyber-attacks known as insider attacks are difficult to tackle because they are perpetrated by trusted users, *i.e.* persons who already have legitimate access to the Information System (IS). These attackers exploit their privileges to perform malicious actions, such as data theft, privilege escalation, or system sabotage. Unlike external threats, insider attacks do not rely on breaching network defenses but instead misuse of authorized access, making prevention less effective. In this paper, authorized access is represented via the Role Based Access Control pattern. In practice, insider threats are not violations of the access control policy, but authorized actions that may lead to unwanted situations. To address this challenge, we developed a formal model-driven framework on top of UML and the B method [1]. The approach is suitable to deal with the dynamic evolution of an IS thanks to the composition mechanism of the B method. The platform, named B4MSecure [10], translates UML and SecureUML [17] models into B and then applies the model-checking facilities of ProB [16] to exhibit malicious scenarios [11].

Detecting an insider attack can be reduced to searching for a specific critical state in the system's state space. If this state is reachable, it means that a sequence of legitimate operations can lead to a security breach. However, a major limitation of model-checking is the combinatorial explosion of states. As systems grow in complexity, the number of possible states increases exponentially, making exhaustive search impractical for real-world applications. A naive brute-force approach would require examining millions or even billions of states, leading to excessive computational costs and long processing times. To circumvent this limitation, we proposed in our previous works [23] a backward symbolic search built on theorem proving and constraint solving. Unfortunately, this approach is not performant because many proof obligations and constraints must be solved interactively. In this paper, we propose to employ a forward search strategy based on ant colony optimization [6], implemented in the API (Ant-based Path Identification) algorithm [7]. Instead of exploring all states blindly [11], like in pure model-checking, or interactively [23], like in symbolic search, API guides the search using a quality-based heuristic, prioritizing paths that are more likely to lead to a given situation.

Ant colony optimization [6,7] (ACO) is a bio-inspired algorithm that mimics the behavior of ants searching for food. In nature, ants deposit pheromones along paths they take, reinforcing routes that lead to successful outcomes. Similarly, in API, artificial ants explore the state space and leave digital pheromones on promising paths. Over multiple iterations, these pheromones accumulate, guiding subsequent ants towards more relevant states. Instead of checking all states, the algorithm prioritizes highly relevant paths, reducing the number of evaluations. The pheromone system allows the algorithm to dynamically adjust and refine its search strategy based on past findings. This work is an attempt to evaluate the efficiency of API for the identification of insider threats, and its capability to find attack scenarios faster than exhaustive search or symbolic search. By combining formal verification with heuristic search, we hope to improve the detection of insider attacks while mitigating the limitations of state-space explosion.

Section 2 presents the B4MSecure platform and the formal modeling of secure IS. Section 3 introduces the API algorithm. Section 4 shows how ProB and API are combined to deal with the insider threat problem and presents the obtained results. Section 5 situates this work within the state of the art. Finally, Section 6 concludes the paper and outlines future work.

2 B for Modeling Secure Information Systems

2.1 Functional and security modeling

To illustrate our approach we consider a simplified model of a bank IS that is inspired by [2]. The UML class diagram of Figure 1 defines functional concerns: customers (class *Customer*) and their accounts (class *Account*). A bank account has a balance (attribute *balance*), an authorized overdraft (attribute *overdraft*) and a unique identifier (attribute *IBAN*). Operations of class *Account* (*e.g. transferFunds*, *withdrawCash* and *depositFunds*) allow one to transfer an amount of money from an account to another, to withdraw and to deposit cash on a given account. A customer is associated with at least one bank account that he does not share. Transferring the ownership of the account or carrying out fraudulent financial operations are examples of insider attacks to prevent. Before searching for these scenarios, it is necessary to define users together with their permissions.

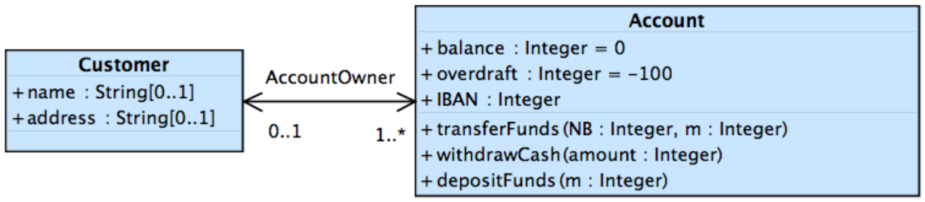
Customer
+name : String[0..1]
+address : String[0..1]
AccountOwner
0..1
1..*
Account
+balance : Integer = 0
+overdraft : Integer = −100
+IBAN : Integer
+transferFunds (NB : Integer, m : Integer)
+withdrawCash (amount : Integer)
+depositFunds (m : Integer)

Fig. 1. Functional *UML* class diagram

Figure 2 is a SecureUML model associated to our class diagram. This model defines two roles: *CustomerUser* and *AccountManager*. They respectively represent the customer of the system and the account manager in charge of the bank's customers.

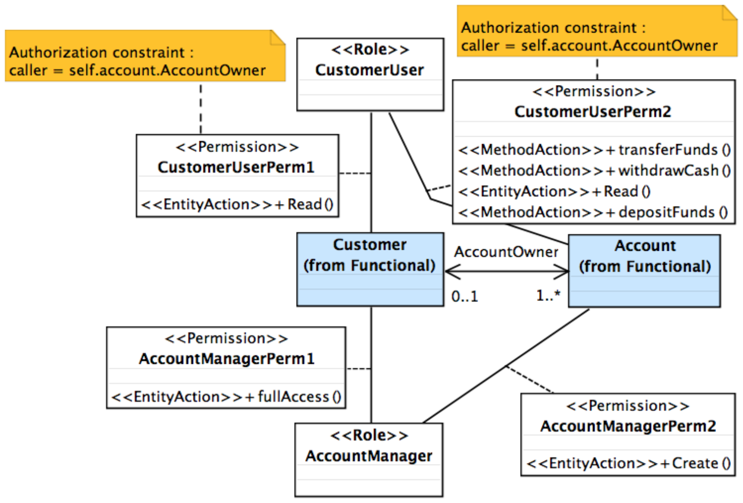
Authorization constraint : caller = self.account.AccountOwner
<<Role>> CustomerUser
Authorization constraint : caller = self.account.AccountOwner
<<Permission>> CustomerUserPerm2
<<MethodAction>>+transferFunds ()
<<MethodAction>>+withdrawCash ()
<<EntityAction>>+Read ()
<<MethodAction>>+depositFunds ()
<<Permission>> CustomerUserPerm1
<<EntityAction>>+Read ()
Customer (from Functional)
AccountOwner
0..1
1..*
Account (from Functional)
<<Permission>> AccountManagerPerm1
<<EntityAction>>+fullAccess ()
<<Permission>> AccountManagerPerm2
<<EntityAction>>+Create ()
<<Role>> AccountManager

Fig. 2. Security modeling with SecureUML

In this IS, customers have the ability to read their personal data (permission *CustomerUserPerm1*) and perform financial operations such as transfer money, deposit, and withdraw cash (permission *CustomerUserPerm2*). Account managers have full access to *Customer* class (permission *AccountManagerPerm1*), allowing them to create, read, and modify customer data. However, their rights on the *Account* class are restricted to the creation of new accounts; which is ensured by permission *AccountManagerPerm2*. An authorization constraint is associated with permissions *CustomerUserPerm1* and *CustomerUserPerm2*, ensuring that only the account holder can perform the corresponding actions on their account. Under this security policy, the account manager has no read or write access to the attributes of the *Account* class.

2.2 Generating B specifications

The translation of a UML class diagram has been discussed in [9] and follows a classical UML-to-B translation. In B, abstract sets represent an abstraction of a set of objects from the real world. As this definition is close to the notion of class in UML, it is used by all UML-to-B approaches to formalize UML classes. Attributes are translated into functions that map the set of existing instances to the attribute's type. The result depends on the attribute's characteristics: mandatory or optional, unique or non-unique, single-valued or multi-valued. For instance, attribute *IBAN* of class *Account* is single-valued, mandatory, and unique ; it is therefore translated into $Account\_IBAN \in Account \rightarrowtail \mathbb{N}$, which is a total injection. Associations are translated similarly into functional relations depending on multiplicities. For example, association *AccountOwner* is translated into a partial surjective function due to multiplicities `0..1` and `1..*`. Figure 3 shows the typing invariants generated by B4MSecure from our class diagram.

INVARIANT
$Account \subseteq ACCOUNT$
$\wedge\ Customer \subseteq CUSTOMER$
$\wedge\ AccountOwner \in Account \mathrel{+\!\!\!\twoheadrightarrow} Customer$
$\wedge\ Account\_balance \in Account \rightarrow \mathbb{Z}$
$\wedge\ Account\_overdraft \in Account \rightarrow \mathbb{Z}$
$\wedge\ Customer\_name \in Customer \nrightarrow STRING$
$\wedge\ Customer\_address \in Customer \nrightarrow STRING$
$\wedge\ Account\_IBAN \in Account \rightarrowtail \mathbb{N}$

Fig. 3. Structural invariants produced by B4MSecure

The SecureUML model is translated by B4MSecure into a B machine that enforces permissions for functional operations based on the roles assigned to a user. For instance, if a user *Paul* is assigned to role *CustomerUser*, he is permitted to read his personal data through the getters of class *Customer*, while other operations such as modification or creation are restricted. The tool produces for every functional operation, a secured operation that verifies (using a security guard) whether the current user is allowed to call the functional operation. The secure operation also verifies the authorization constraints, if they are defined in the underlying permissions, and updates the assignment of roles when required.

Figure 4 shows the secure operation associated to *Account_transferFunds*. The security guard is defined in clause *SELECT*. It verifies that the functional operation belongs to set *isPermitted*[*currentRoles*], where definition *currentRoles* refers to the roles activated by *currentUser* (in a session) as well as their super-roles. The security guard of *secure_Account_transferFunds* is strengthened with the authorization constraint of permission *CustomerUserPerm2*. For a permission p associated to a role r and a constraint c, the tool adds guard $(r \in \mathit{currentRoles} \Rightarrow c)$ to all operations that are concerned with p. In this case the constraint is: $\mathit{AccountOwner}(\mathit{aAccount}) = \mathit{currentUser}$.

```
secure_Account_transferFunds(aAccount, NB, m) =
PRE
    $aAccount \in Account \wedge NB \in \mathbb{N} \wedge m \in \mathbb{N}_1$ [$\wedge \ldots$]
THEN
  SELECT
    $Account\_transferFunds\_ \in isPermitted[currentRoles]$
    $\wedge (CustomerUser \in currentRoles \Rightarrow AccountOwner(aAccount) = currentUser)$
  THEN
    Account_transferFunds(aAccount, NB, m)
  END
END;
```

Fig. 4. Secured operation

The B specifications generated by B4MSecure from a given class diagram are designed to be animated using ProB [16]. This enables the observation of the IS's evolution and the impact of execution scenarios on its functional state. B4MSecure generates all basic operations, including the creation/deletion of class instances, creation/deletion of links between instances, and getters/setters for attributes and links. Integrity constraints can be introduced using B invariants and proof of correctness for invariant preservation can be done via a proof assistant such as AtelierB.

2.3 Malicious behaviors

To identify malicious behaviors, we analyze the set of finite observable traces of our B specification. The latter can be represented using trace semantics, which consist of an initialization substitution *init*, a set of operations $\mathcal{O}$, and a set of state variables $\mathcal{V}$. A functional behavior is an observable sequence $\mathcal{Q}$ defined as:

$$\mathcal{Q} \mathrel{\hat{=}} \mathit{init} \ ; \ op_1 \ ; \ op_2 \ ; \ \ldots \ ; \ op_m$$

such that $\forall i.(i \in 1..m \Rightarrow op_i \in \mathcal{O})$ and there exists a sequence $\mathcal{P}$ of states that do not violate invariant properties:

$$\mathcal{P} \mathrel{\hat{=}} s_0 \ ; \ s_1 \ ; \ \ldots \ ; \ s_{m-1}$$

where s_0 is an initial state, and each op_i is enabled from state s_{i-1}, leading to state s_i. $\mathcal{P}$ is called a path.

The security model filters functional behaviors by analyzing access control premises, which are triplets (u, R, c) where u is a user, R is a set of roles assigned to u, and c is an authorization constraint. An observable secure behavior is a sequence $\mathcal{Q}$, where for every step i, the premise (u_i, R_i, c_i) is valid (expressed as $(u_i, R_i, c_i) \models true$). This means that the roles R_i activated by user u_i grant the right to execute operation op_i, and any constraint c_i must be satisfied. The following sequence of premises must be valid for $\mathcal{Q}$:

$$(u_1, R_1, c_1) \ ; \ (u_2, R_2, c_2) \ ; \ \ldots \ ; \ (u_m, R_m, c_m)$$

The search for a malicious scenario reduces to finding a specific state in the state space. The path from the initial state to this target state represents the sequence of operations required to execute the attack. Therefore, a malicious behavior, done by a user u, considering an access control policy, is an observable path $\mathcal{P}$ with m steps such that:

- op_m is a critical operation associated with an authorization constraint c_m.
- State s_{m-1} is called malicious and enables op_m.
- User u is malicious and aims to execute op_m by exploiting his roles R_u.
- s_0 is an initial state where $(u, R_u, c_m) \models false$.
- For every step i ($i \in 1..m$), the premise $(u, R_u, c_i) \models true$.

In other words, malicious user u is not initially allowed to execute the critical operation, but he is able to run a sequence of operations leading to a state from which he can execute this operation. Suppose that s_0 is as follows:

$$\boxed{\begin{array}{l} s_0 \mathrel{\hat{=}} \\ \quad Account = \{cpt_1, cpt_2\} \\ \quad Customer = \{Paul, Martin\} \\ \quad Account\_balance = \{(cpt_1 \mapsto 300), (cpt_2 \mapsto -100)\} \\ \quad AccountOwner = \{(cpt_1 \mapsto Paul), (cpt_2 \mapsto Martin)\} \\ \quad Account\_IBAN = \{(cpt_1 \mapsto 111), (cpt_2 \mapsto 222)\} \\ \quad Account\_overdraft = \{(cpt_1 \mapsto -100), (cpt_2 \mapsto -100)\} \end{array}}$$

In this state, *Paul* is a customer and owns account cpt_1 with a balance of 300. *Bob*, as an *AccountManager*, cannot execute operations like *transferFunds* or *withdrawCash* on cpt_1. A static query such as "Is Bob able to transfer funds from Paul's account?" would return NO, since the permission granted to a manager on the *Account* class only allows instance creation. The more pertinent question is, "Is there a sequence of operations that Bob can execute to gain the ability to transfer funds from Paul's account?". To answer this, one naive approach is to use ProB's model-checking feature to exhaustively explore the state space and find states that satisfy property: $AccountOwner(cpt_1) = \text{Bob}$. We are looking for a sequence of operations executed by Bob that allows him to become the owner of cpt_1, thereby granting him the permission to execute an action he initially could not. Sequence $\mathcal{Q}$ that we want to exhibit is represented in Figure 5.

The attacker (Bob) creates a fictitious customer profile and associates it with a newly created bank account. To gain control over an existing customer account (cpt_1), Bob first assigns another dummy account (cpt_4) to Paul. Then, Bob removes the ownership link between Paul and cpt_1, and assigns cpt_1 to himself.

```
/* step 1: create customer Bob */
    Connect(Bob, {AccountManager}) ;
    setCurrentUser(Bob) ;
    secure_Account_NEW(cpt3, 333) ;
    secure_Customer_NEW(Bob,{cpt3}) ;
/* step 2: get the ownership of Paul's Account */
    secure_Account_NEW(cpt4, 444) ;
    secure_Customer_AddAccount(Paul,{cpt4}) ;
    secure_Customer_RemoveAccount(Paul,{cpt1}) ;
    secure_Customer_AddAccount(Bob,{cpt1}) ;
/* step 3: attack */
    disConnect(Bob) ;
    Connect(Bob, {CustomerUser}) ;
    secure_Account_transferFunds(cpt1, 333, 100) ;
```

Fig. 5. Malicious scenario

Bob logs in the system as a customer in order to transfer funds from cpt_1. In this example, ProB reached a timeout after exploring millions of transitions, indicating that the state space is too large to be explored efficiently. To solve this issue, we proposed in [11] a CSP||B that helps ProB searching in a subset of the state space. ProB was able to exhibit the sequence above after exploring about 70,000 states and 200,000 transitions. We also proposed a more generic technique based on a backward symbolic search [23]. This approach follows a two-step process: it first identifies symbolic functional executions that could potentially lead to a malicious scenario using theorem proving, and then checks their feasibility against the security model using constraint solving. While this method is particularly useful for understanding the origin and structure of attack scenarios, its main drawback lies in its low performance and interactive nature, as many proofs and constraints require manual intervention. In the example discussed, this approach required 38 minutes to identify the attack sequence.

3 Ant Colony Optimization – ACO

Ant colony algorithms [6,7] are a class of heuristic search algorithms inspired by the foraging behavior of ants. Developed in the 1990s, these algorithms have proven to be highly effective in solving combinatorial optimization problems, where exhaustive search methods are often computationally prohibitive. Different species of ants exhibit various behaviors, which have been modeled into a range of algorithms with different objectives.

3.1 Ant-based Path Identification with Pachycondyla apicalis

This algorithm has been formalized by N. Monmarché in 2000 [21,20]. It is inspired by *Pachycondyla apicalis*, a species of ants present in South America. In most ant colony algorithms, the global solution is the one that attracts the most individuals, this is called *stigmergy.* The search space is divided into several regions, each of which is explored by a group of ants. The ants of a group are attracted by the pheromones left by their congeners. The pheromones are a way to communicate the quality of the solutions found. The more a region is visited, the more the pheromones are deposited and the more the region is attractive. The algorithm is iterative: ants explore regions, pheromones are deposited, regions are updated and ants are attracted to regions with most pheromones.

The particularity of API species is that they live in small colonies (a few dozen individuals), with an unstable habitat that is likely to evolve often, and little direct communication between individuals. These characteristics have been transcribed into the algorithmic that we are experimenting in this work. The colony sends n ants a_1, ..., a_n (called *foragers*) around the nest. Each ant creates and then stores in memory p hunting sites. In parallel with the other ants, the ant randomly chooses one of its hunting sites, denoted S, and begins a local exploration. If the exploration is successful, it replaces S with its new hunting site S'. Otherwise, it counts an additional failure for the site S. If the number of failures of S exceeds a threshold called *local patience*, it is forgotten and replaced by a new random site. Patience allows to dig a track for a few turns rather than abandoning it immediately. The colony regularly recalls its foragers and looks at their results. If a hunting site of an ant is better than those of the other ants and the current position of the nest, the whole nest moves and the memory of all the ants starts from scratch. The procedure then starts again: the ants are scattered around the nest again.

3.2 Algorithm

The *API* algorithm uses the following notations:

- $\mathcal{S}$ is a search space;
- $N \in \mathcal{S}$ is the position of the nest, initialized with the random operator $\mathcal{O}_{\text{rand}}$;
- $\mathcal{O}_{\text{explo}}$ returns a point S' in the neighborhood of a point $S \in \mathcal{S}$;
- $f : \mathcal{S} \to \mathbb{R}$ is the objective function to minimize;
- $n_s(a_i)$ is the number of hunting sites of ant a_i;
- A_{site} and A_{locale} are the maximum amplitudes to create hunting sites;
- e_j is the number of failures associated with hunting site s_j for a given ant;
- P_{locale} is the threshold of failures before an ant abandons a site.

The algorithm is presented in Figure 6. First, it chooses the nest location N using an operator $\mathcal{O}_{\text{rand}}$, which represents a random initialization function. A variable T is initialized to keep track of the number of explorations and the loop runs until the stopping condition is reached.

The stopping condition can be based on a maximum number of iterations, convergence to an optimal solution, or performance metrics, etc. Each ant performs an exploration process (API-FORAGING) searching for better solutions and using pheromone updates. If a better solution (S^+) is found, the nest is moved to this new location and the memory of all ants is cleared so that they

Algorithm 1: API()

```
1  Choose the initial nest location : $N := \mathcal{O}_{\text{rand}}$;
2  $T := 0$; // Number of ant explorations
3  while The stopping condition is not met do
4      foreach ant $a_i$ do
5          API-FORAGING($a_i$);
6      if The nest must be moved then
7          $N := S^+$; // Best solution found by an ant
8          Clear the memory of all ants;
9      $T := T + 1$
10 return $(S^+, f(S^+))$
```

Fig. 6. API Main Algorithm

can start fresh from the new nest. The algorithm returns the best solution found (S^+) along with its objective function value ($f(S^+)$). Algorithm of Figure 7 simulates the behavior of a single ant a_i in the search space. The ant has a memory of p hunting sites. If the memory is not full, the ant creates a new hunting site around the nest. Otherwise, it explores the neighborhood of the last site visited. If the exploration is successful, the ant moves to the new site. Otherwise, it counts a failure and may abandon the site if the number of failures exceeds a threshold. The algorithm returns the best solution found by the ant.

3.3 Discussion

API has been applied in many optimization problems such as the traveling salesman problem, the quadratic assignment problem, and the job-shop scheduling problem. It has also been used in various applications such as data mining [18], image processing [8], and network routing [5]. In this work we use it to search for malicious scenarios in the execution of a formal model of an IS. The application of the algorithm requires only the knowledge of the exploration operators $\mathcal{O}_{\text{explo}}$ and $\mathcal{O}_{\text{rand}}$. The exploration operator $\mathcal{O}_{\text{explo}}$ is specific to the problem to be solved. It is a function that generates a new solution in the neighborhood of a given solution. The random operator $\mathcal{O}_{\text{rand}}$ is used to initialize the search space.

In our opinion, *API* is a good solution to identify insider threats because the behavior of the ant colony is somehow close to that of an attacker. Attackers first try to identify critical states of the software and then he/she concentrates his/her efforts to find security flaws starting from these critical states. Similarly, API ants focus on the most promising sites to find the best possible solution. Both API ants and attackers adopt an incremental, exploratory approach. Indeed, API ants begin by randomly exploring potential paths, but over time, they reinforce the most promising routes based on pheromone trails (here function f). Insiders follow a similar strategy: they first explore the system, looking for weaknesses; once they find a critical state (*e.g.,* a vulnerability or misconfiguration), they focus their efforts on exploiting it. Furthermore, in API, ants are designed to

Algorithm 2: API-FORAGING(a_i)

```
if $n_s(a_i) < p$ then
    // The ant's memory is not full
    $n_s(a_i) := n_s(a_i) + 1$;
    Creation of a foraging site around the nest: $s_{n_s(a_i)} := \mathcal{O}_{\text{explo}}(N, A_{\text{site}})$;
    Initialization of the failure counter for the site: $e_{n_s(a_i)} := 0$;
else
    Let $s_j$ be the last site explored by the ant ;
    if $e_j > 0$ then
        // The last exploration was unsuccessful.
        Randomly select a foraging site $s_j (j \in \{1, \ldots, p\})$
    Local exploration around $s_j$ : $s' := \mathcal{O}_{\text{explo}}(s_j, A_{\text{locale}})$ ;
    if $f(s') < f(s)$ then
        $s_j := s'$;
        $e_j := 0$;
    else
        $e_j := e_j + 1$;
        if $e_j > P_{locale}$ then
            Delete site $s_j$ from the ant's memory ;
            $n_s(a_i) := n_s(a_i) + 1$;
```

Fig. 7. Simulation of a given ant a_i

focus on highly probable paths that lead to optimal solutions, much like how real ants reinforce the best paths to a food source. In cybersecurity, attackers prioritize high-value targets, such as privileged data (*e.g.*, account cpt_1). They start from accessible points and progressively refine their approach. Finally, API ants adjust their search behavior dynamically, reacting to previous successes. Attackers do the same by adapting their attack strategies based on the system's defenses, security policies, and feedback from failed attempts.

4 API and ProB to exhibit insider threats

The API algorithm is a generic algorithm. It only specifies how the ants are organized. Here, we propose a modeling of our optimization problem for the API algorithm based on the state space discovered by ProB.

4.1 State evaluation functions

The *state evaluation function*, denoted f, formalizes what is an "interesting state" of the state space with regard to an insider attack. This function can be a binary function that checks if a current state strictly approaches the malicious one. A drawback is that a path can stagnate on few states or slightly regress before being considered interesting. An evaluation in $\{0, 1\}$ rejects all states that do not advance. Our idea is to have a refined evaluation function. In the API algorithm, the local patience of the ants allows to visit states that are not immediately fruitful, that's why we propose an evaluation function f with values in $[0, 1]$.

Considering our formal B specification, $\mathcal{S}$ is simply the state space discovered by ProB during the exploration process. We denote by s^* the malicious state and by $s(v)$ the value of variable v ($v \in \mathcal{V}$) in a given state s. Let $w_v \in \mathbb{R}$ be a weight associated with variable v to ponder its impact on the exploration. We define the evaluation function as:

$$f : \mathcal{S} \to [0, 1]$$
$$f(s) = \frac{\sum_{v \in \mathcal{V}} w_v \cdot \delta(s^*(v), s(v))}{\sum_{v \in \mathcal{V}} w_v}$$

Function f is an objective function. It defines a weighted average of the distance between the current values of the variables (in state s) and the desired ones (in state s^*). Intuitively, a desired path $\mathcal{P}$ is a sequence of states whose variables look more and more like those of the target state, until they match completely. In our approach, the malicious state s^* is a state that enables a critical operation (*e.g.*, *transfer_ Funds*) and satisfies security premises of the attacker (*i.e.*, Bob). Note that we generate state s^* using the constraint-solving feature of ProB, with various strategies not detailed here for space reasons. State s^* is:

$$\begin{aligned}
s^* \;\hat{=}\; & \\
& \mathit{Account} = \{cpt_1, cpt_2, cpt_3, cpt_4\} \\
& \mathit{Customer} = \{Paul, Martin, Bob\} \\
& \mathit{Account\_balance} = \{(cpt_1 \mapsto 300), (cpt_2 \mapsto -100), (cpt_3 \mapsto 0), (cpt_4 \mapsto 0)\} \\
& \mathit{AccountOwner} = \{(cpt_1 \mapsto Bob), (cpt_2 \mapsto Martin)(cpt_3 \mapsto Paul), (cpt_4 \mapsto Bob)\} \\
& \mathit{Account\_IBAN} = \{(cpt_1 \mapsto 111), (cpt_2 \mapsto 222), (cpt_3 \mapsto 333), (cpt_4 \mapsto 444)\} \\
& \mathit{Account\_overdraft} \\
& \qquad = \{(cpt_1 \mapsto -100), (cpt_2 \mapsto -100), (cpt_3 \mapsto -100), (cpt_4 \mapsto -100)\}
\end{aligned}$$

A state s_i where $\mathit{Account} = \{cpt_1, cpt_2, cpt_3\}$ is less distant from s^* than a state s_j where $\mathit{Account} = \{cpt_4\}$. This distance between s^*, s_i and s_j is measured with the evaluation function f. The similarity δ between two values is defined according to the type of the variable. The objective is to minimize f as much as possible and if $f(s) = 0$, then s is the target state. To this purpose we apply the following measures, which are adapted from Jaccard's Index of Similarity [12]:

– For a variable derived from a class or an association, we measure the distance between two sets (A and A). Specifically, we compute 1 minus the ratio of the size of their intersection to the size of their union. As a result, $\delta(A, A) = 0$ when $A = A$, and $\delta(A, A) = 1$ when $A \cap A^* = \emptyset$.

$$\delta(A^*, A) = 1 - \frac{\mathrm{card}(A \cap A^*)}{\mathrm{card}(A \cup A^*)}$$

– For an integer variable, the comparison is bounded in an interval. The distance is defined as the difference between the two values divided by the amplitude of the interval. Interval $[A_{\min}, A_{\max}]$ is chosen such that it covers all values of A and A^*.

$$\delta(A^*, A) = 1 - \frac{|A - A^*|}{|A_{\min} - A_{\max}|}$$

– For a boolean variable, the distance is 0 if the variable values are equal and 1 otherwise.

$$\delta(A^*, A) = \begin{cases} 0 \text{ if } A = A^* \\ 1 \text{ if } A \neq A^* \end{cases}$$

Function f computes a weighted average. Indeed, weights $w_v \in \mathbb{R}$ depend on the search strategy. By default, they are equal to 1. They are useful to limit the impact of some variables compared to others during the search. For example, a transition leading to a state where δ becomes 0 for a boolean variable decreases f much more than the inclusion of a new value for a class variable of type set. This can unbalance the search of the ants. One solution is to reduce the weight of the boolean variable in comparison with a class variable.

4.2 Search heuristics

The movement of ants requires two operators: $\mathcal{O}_{\text{rand}}$ to give an initial location to the nest; and $\mathcal{O}_{\text{explo}}$ to guide the ants in the search space. Since the search space $\mathcal{S}$ is that of ProB, a natural choice for the nest is the root of the state space. This choice removes randomness in the initial position of the nest.

Operator $\mathcal{O}_{\text{explo}}$ is a heuristic for moving in the search space. It generates a point s' in the neighborhood of a point $s \in \mathcal{S}$. Here, a point is a state explored by the *model-checker*. The neighborhood corresponds to all the states that can be reached in one transition from a given state. A chosen solution for $\mathcal{O}_{\text{explo}}$ is to return a random transition among those evaluated by `ProB`. All transitions have the same probability of being chosen. Since transitions go in one direction, ants may get stuck in a sub-search space that corresponds to a local minimum. To address this issue, we add an artificial "backtracking" transition. Figure 8 represents a simplified `ProB` state space. Red arrows are transitions between states already explored by the ants. Dashed red arrows represent the backtracking transitions that allow the ant to go back. Blue nodes are states that become accessible to the ant due to the backtracking transitions. This mechanism leads to the creation of a spanning tree for the already visited states.

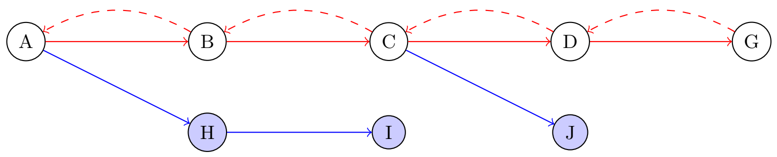

Fig. 8. Backtracking activation on a simplified state space of *ProB*.

The *API* algorithm does not define any condition on the number of explorations. At any time during execution, it may provide a valid solution in $\mathcal{S}$. To stop it, several strategies are possible. One may define a fixed number of explorations $T_{\max}$, meaning as long as $T < T_{\max}$, a new exploration is started. This solution has the disadvantage of not taking into account the number of ants. Thus, for the same $T_{\max}$, an algorithm with ten ants will take twice as long to execute as an algorithm with five ants. An other idea is to stop when the ants find a solution that no longer improves. If the current solution does not change after a certain number of explorations, the algorithm stops. In the same idea one can stop when the algorithm reaches a solution *close enough* to the optimal ($f(S^+) < \varepsilon$). This corresponds to the case where state S^+ is close to s^* without being identical. If only a few operations are missing to reach the optimal state, a *deterministic* search algorithm can take over. This stopping condition should only be triggered after a certain number T of iterations. Otherwise, *API* will never reach the optimal state.

Another solution would be to define a maximum number of visited states. In this solution, a common counter for all ants is added and when a state is visited for the first time, the counter is incremented. An advantage of this solution is that the execution time is less dependent on the number of ants. Also, the number of visited states is an important criterion in *model-checking* algorithms. `ProB` includes this data when animating a model. This choice therefore facilitates comparisons between algorithms. For the example of this paper the stopping criterion we use is the maximum number of state evaluations (in which a distance is computed). As for the maximum number of visited states, a global counter is added. We count the number of times a state is evaluated, whether for the first time or not. The execution time is independent of the number of ants, making it a good performance measure. Moreover, it is a relevant measure for animating models in *ProB*. Indeed, evaluating a state is a very costly operation. It represents 60% of the computation time in the implementation of *API*. The δ functions presented in section 4.1 are first translated into the B syntax, and then sent to `ProB` during exploration, evaluated, and returned back to the algorithm. This criterion is combined with an option that allows the algorithm to stop immediately if the final state is found ($f(s) = 0$). The state count is kept to compare various executions of the *API* algorithm.

The nest movement criterion adjusts the frequency at which the colony recalls its ants to check their solutions. If this delay between explorations is too long, the ants may forget satisfactory solutions. Conversely, if it is too short, the ants do not have time to explore the search space and the colony does not progress. Based on local search, it is possible to use a patience factor for the nest, denoted P_N. Let p be the size of an ant's memory and P_{locale} its local patience. In [20], the nest's patience is: $P_N = 2 \times (P_{\text{locale}} + 1) \times p$. The evaluation function f can have significant variations with a few number of variables. For the example in this article, we adapt the initial formula with: $P_N = (P_{\text{locale}} + 1) \times p$.

Amplitudes define the neighborhood area of a state during exploration. The global amplitude, denoted A_{site}, provides an area to place the ants around the nest at the beginning of the exploration. The local amplitude, denoted A_{locale}, provides an area where the ants choose their hunting sites. The amplitude is normally defined as a proportion relative to the search space ($A \in [0, 1]$). Since the state space is discovered by ProB during exploration, its size is not known in advance. Therefore, we represent the amplitude by a maximum number of transitions ($A \in \mathbb{N}$) between the starting state and the state to be explored. Our experiments apply the following parameters:

- Number of ants: $n = 5$;
- Ant memory: $p = 3$;
- Amplitudes (local and global): $A_{\text{locale}} = 3$, $A_{\text{site}} = 15$;
- Patience (local and global): $P_{\text{locale}} = 3$, $P_{\text{N}} = (P_{\text{locale}} + 1) \times p = 12$;
- Maximum number of evaluations: 1000;

4.3 Results and discussion

The *API* algorithm is non-deterministic, as its efficiency depends on the random exploration performed by the ants in the state space. Consequently, the number of states visited before identifying a critical state can vary significantly between executions. Table 1 presents the results of 20 independent executions of the *API* algorithm on the example of this paper.

Run	**Nb of evaluations**	**Length of critical path**	**Transitions (total)**	**Transitions (unique)**	**States**	**Time (s)**
1	322	21	764	442	135	41.69
2	214	22	477	295	81	27.35
3	154	23	524	279	55	23.77
4	423	24	1006	708	187	59.57
5	323	22	821	564	158	51.58
6	426	29	1089	522	140	46.91
7	406	28	844	601	188	57.74
8	246	24	517	311	95	31.20
9	176	19	421	266	80	26.17
10	284	17	672	456	125	41.21
11	302	25	748	508	137	46.49
12	566	25	1270	900	242	75.30
13	282	20	761	490	111	39.65
14	298	20	772	523	139	46.30
15	420	19	1054	655	168	54.59
16	318	22	715	499	147	45.67
17	308	18	804	506	125	41.86
18	413	19	912	629	182	56.36
19	393	23	818	511	164	49.27
20	286	23	683	452	124	40.15
	Avg: 321.4	**21.6**	**782.0**	**518.8**	**134.7**	**44.62**

Table 1. Summary of 20 runs.

These results are fully reproducible. The implementation, along with the B models and automation scripts used in the experiments, is open-source and publicly available at https://github.com/meeduse/insider_threats. Each row corresponds to a complete run of the algorithm. The table reports: the number of calls to the evaluation function f (even when a state is visitied several times), the length of the reconstructed critical path, the total and unique number of transitions executed, the total number of visited states, and the total execution time in seconds. The last row shows the average value for each metric over the 20 runs. The execution traces reveal that the algorithm's performance may fluctuate, but remains consistently effective. In 100% of these executions, API successfully reconstructs a path reaching the expected critical state.

Compared to exhaustive methods such as pure model-checking or the CSP||B approach [11], API demonstrates a significant reduction in the number of visited states. On average, only 321 states are explored per run to find the attack scenario, whereas CSP||B required up to 70,000 states and 200,000 transitions. We have also applied our symbolic search approach [23] to this simple example. Since the method relies on symbolic states rather than concrete ones, it is difficult to compare it directly using criteria such as the number of states or transitions. However, in terms of performance, the symbolic search proved inefficient, requiring 38 minutes to discover a single attack scenario, whereas the API-based approach achieved the result in an average of 44 seconds. This shows the benefit of API's exploration strategy, which leverages heuristic search instead of systematic enumeration or constraint solving and symbolic proofs. The length of the critical path also varies, ranging from 17 to 29 actions. The minimal length of 17 (highlighted in Table 1) is close to the optimal path of 11 operations identified manually. On average, 22 actions are needed to reach a critical state.

These results support the conclusion that API has the potential to outperform traditional approaches by substantially reducing state-space exploration and improving overall execution time. We experimented our approach on many case studies: Medical IS [14], Meeting scheduler [3], Bank IS [2], Conference Review [25]. API is more than 500x more efficient than pure model-checking. However, the algorithm's random nature introduces variability in the number of steps required to reach the critical state. Furthermore, API parameters are crucial in obtaining good results but there are no predefined rules for choosing their values. Finding an optimal set of parameter values requires extensive testing. The selected values provided a good balance between exploration and convergence in our case study. In particular, setting the number of ants to $n = 5$ allowed sufficient coverage of the state space while keeping the number of evaluations per ant high enough for each to reach relevant areas. Interestingly, increasing the number of ants does not always improve performance. For instance, setting $n = 50$ while keeping the evaluation limit at 1000 drastically reduces the number of evaluations available to each ant. In such settings, none of the ants may explore deeply enough to discover the critical path. Finding the optimal parameter configuration remains an open question. Our experiments show that performance improves when the number of ants is increased moderately. For example, using $n = 10$ ants yields better results than $n = 5$, without compromising the depth of exploration too severely. More systematic parameter tuning could further enhance the algorithm's efficiency.

5 Related works

As far as we know, works that addressed access control together with a formal method did not deal with the insider threat problem such as discussed in this paper. However, we can assume that this kind of threat is a typical reachability problem. In [25], the authors proposed a plain model-checking approach built on security strategies. A similar approach is proposed in [13] to validate access control in web-based collaborative systems. Even though their experiments show that they achieve better results compared to [25], the approach still has a partial coverage of realistic policies. In [19], the authors proposed two approaches to prove reachability properties in a B formal information system modelling, but unlike our work, they do not search sequences leading to a goal state.

Ant Colony Optimization has also been successfully applied to software testing problems. Several surveys explore this connection, including the work of Suri and Singhal [24], which provides a classification of ACO-based techniques for test case generation, test suite minimization, and prioritization. More recent surveys, such as [4], further expand on the role of ACO and other metaheuristics in automated test data generation. While these applications share with our approach the principle of guided search in large input spaces, they differ in their objectives and representations: our work focuses on local exploration guided by a Jaccard-based distance metric, derived from formal specifications and evaluated dynamically using the ProB model checker.

Another relevant line of research concerns directed model checking. In the context of the B method and ProB, Leuschel and Bendisposto [15] introduced heuristic-guided search strategies to steer the exploration toward goal states. The ProB tool supports this approach through the `HEURISTIC_FUNCTION` feature, which allows users to define custom heuristics to influence transition prioritization during state space exploration [22]. Although this technique shares with our ant-based method the use of heuristic guidance, the two approaches differ fundamentally in their mechanisms. API is a population-based, stochastic metaheuristic that relies on memory and pheromone reinforcement to guide exploration over time. In contrast, directed model checking in ProB follows a deterministic or best-first strategy driven by a fixed heuristic function—either statically defined or dynamically computed for each state.

6 Conclusion

Insider threats represent a major challenge in cybersecurity due to the legitimate privileges and trust inherently granted to internal users. Formal methods, and in particular model checking, have long been used to rigorously analyze system behavior. However, the state-space explosion problem often limits their scalability, especially for realistic or evolving information systems. This paper shows that combining formal verification with ant colony optimization offers a promising direction for identifying malicious scenarios more efficiently.

We proposed an original method based on the API variant of ACO [21], applied to formal models expressed in the B method and analyzed using the ProB model checker [16]. To our knowledge, this is the first application of ACO to the insider threat detection problem in a formal setting. Our experiments confirm that the API algorithm can guide the exploration toward critical states using a fitness function tailored to a target configuration. Compared to exhaustive or symbolic search, our approach significantly reduces the number of visited states while still reconstructing full attack paths.

Beyond the specific case study, the proposed method is more generally applicable. The algorithm can be extended to handle more complex attacker models, including coalitions of insiders. In this setting, the search would aim to discover coordinated sequences of actions performed by multiple users that lead to policy violations. Regarding the specification of critical states, our current implementation assumes a known target state (e.g., a policy breach where a security constraint is violated). Another approach could be to detect when an authorization constraint becomes unsatisfied. Such dynamic assertion checking could serve as a trigger to automatically mark a state as "critical".

Future work will focus on these extensions: (1) integrating collaborative attacker models and verifying multi-user attack strategies, (2) exploring automatic generation of target states based on the negation of authorization constraints, and (3) improving the scalability of the algorithm through parallel search and adaptive pheromone strategies. We believe this opens a new path for bridging formal policy analysis and heuristic-driven attack simulation.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. A. Bandara, H. Shinpei, J. Jurjens, H. Kaiya, A. Kubo, R. Laney, H. Mouratidis, A. Nhlabatsi, B. Nuseibeh, Y. Tahara, T. Tun, H. Washizaki, N. Yoshioka, and Y. Yu. *Security Patterns: Comparing Modeling Approaches.* IGI Global, 2010.
3. D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information & Software Technology*, 51, 2009.
4. M. Chhillar and R. Bhatia. A systematic literature review on metaheuristic test data generation techniques. *ACM Computing Surveys*, 54(2):1–37, 2021.
5. F. Debbat and F. T. Bendimerad. Assigning cells to switches in cellular mobile networks using hybridizing api algorithm and tabu search. *International Journal of Communication Systems*, 27(12):4028–4037, 2013.
6. M. Dorigo, M. Middendorf, and T. Stützle, editors. *ANTS'2000 - From Ant Colonies to Artificial Ants: Second International Workshop on Ant Algorithms*, Brussels, Belgium, September 8-9 2000.
7. M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, Cambridge, MA, 2004.

8. A. M. Hannane and H. Fizazi. Supervised images classification using metaheuristics. *Computer Modelling & New Technologies*, 20(3):17–23, 2016.
9. A. Idani. The B method meets MDE: review, progress and future. In R. S. S. Guizzardi, J. Ralyté, and X. Franch, editors, *16th International Conference on Research Challenges in Information Science (RCIS'22)*, volume 446 of *LNCS*, pages 495–512, Spain, 2022. Springer.
10. A. Idani and Y. Ledru. B for modeling secure information systems - the b4msecure platform. In M. J. Butler, S. Conchon, and F. Zaïdi, editors, *17th International Conference on Formal Engineering Methods*, volume 9407 of *LNCS*, pages 312–318. Springer, 2015.
11. A. Idani, Y. Ledru, and G. Vega. A process-centric approach to insider threats identification in information systems. In *18th International Conference on Risks and Security of Internet and Systems (CRiSIS'23)*, volume 14529 of *LNCS*, pages 231–247. Springer, 2023.
12. P. Jaccard. The probabilistic basis of jaccard's index of similarity. *Systematic Biology*, 45(3):380–385, 1996.
13. M. Koleini and M. Ryan. A knowledge-based verification method for dynamic access control policies. In *13th International Conference on Formal Engineering Methods, ICFEM*, volume 6991 of *LNCS*, pages 243–258. Springer, 2011.
14. Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, and M.-A. Labiadh. Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design (IJISMD)*, 2014.
15. M. Leuschel and J. Bendisposto. Directed model checking for b: An evaluation and new techniques. In *SBMF 2010: Formal Methods*, volume 6527 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2010.
16. M. Leuschel and M. Butler. Prob: an automated analysis toolset for the b method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, Mar 2008.
17. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, London, UK, UK, 2002. Springer-Verlag.
18. G. C. Luh and C. Y. Lin. Optimal design of truss structures using ant algorithm. *Structural and Multidisciplinary Optimization*, 36(4):365–379, 2008.
19. A. Mammar and M. Frappier. Proof-based verification approaches for dynamic properties: application to the information system domain. *Formal Asp. Comput.*, 27(2):335–374, 2015.
20. N. Monmarché. *Algorithmes de fourmis artificielles : applications à la classification et à l'optimisation*. PhD thesis, Université François Rabelais, Tours, 2000.
21. N. Monmarché, G. Venturini, and M. Slimane. On how pachycondyla apicalis ants suggest a new search algorithm. *Future Generation Computer Systems*, 16(8):937–946, 2000.
22. ProB. Tutorial: Directed model checking. https://prob.hhu.de/w/index.php?title=Tutorial_Directed_Model_Checking. Accessed: 2025-04-09.
23. A. Radhouani, A. Idani, Y. Ledru, and N. B. Rajeb. Symbolic search of insider attack scenarios from a formal information system modeling. *LNCS Transactions on Petri Nets Other Models of Concurrency*, 10:131–152, 2015.
24. B. Suri and S. Bawa. Literature survey of ant colony optimization in software testing. *International Journal of Computer Applications*, 45(14):1–6, 2012.
25. N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.