5-2024

Applications of Artificial Intelligence to Information Privacy

James Gunder Frazier '24
Hamilton College

HAMILTON COLLEGE SENIOR FELLOWSHIP PROGRAM

APPLICATIONS OF ARTIFICIAL INTELLIGENCE TO INFORMATION PRIVACY

JAMES GUNDER FRAZIER
HAMILTON COLLEGE SENIOR FELLOW

Project submitted to the Faculty of Hamilton College in partial fulfillment of the requirements for the degree, Bachelor of Arts

2024

Certification of approval:

Principal Advisor

Co-Advisor

Applications of Artificial Intelligence to Information Privacy: Using a Two-Phase Evolutionary Designer to Configure Software Defined Perimeters

James Gunder Frazier
Hamilton College
Clinton, New York, USA
jgfrazie@hamilton.edu

ABSTRACT

Software Defined Perimeter (SDP) is a zero-trust network-isolation defense technique which aims to limit security risks by giving dynamic account type assignments to network users. Despite SDP being proven as an effective defense strategy in various domains, it has yet to see wide-spread use due to its drawbacks. One of SDP's most pressing issues is the need for an expert to manually configure it for each unique application. Here we describe a novel system for designing SDP networks called SDPush which can automatically design and analyze possible configurations for a given network with user-specifications. Since there is not a systematic approach for account type design and assignment, we develop a two-step optimization system consisting of a bitstring genetic algorithm and a genetic programming sub-system for designing and evaluating SDP networks respectively. In order to evolve an SDP configuration exhibiting the user-specified characteristics while also minimizing security risk, we implement our system to support multi-objective search spaces by providing the system's training set with different cases aimed at evaluating different aspects of the network configuration. We present initial results of experiments on networks of varying size and characteristic requirements.

CCS CONCEPTS

• **Security and Privacy** → **Vulnerability Management**; • **Evolutionary Computation**;

KEYWORDS

evolutionary computation, decision making, cybersecurity, networks

1 INTRODUCTION

In 2023, 3205 data records were compromised with each resulting in a net-loss of approximately 4.45 million USD on average [1, 2]. Every year the severity and frequency of these data breaches increases. These data breaches can occur from intentional means e.g. malware intrusions, or from inadvertent means e.g. the accidental publication of sensitive documents or information [4]. Recently, network defense techniques have aimed to 1) mitigate the frequency and severity of such events, 2) maintain applicability to various use-cases, and 3) affect user-experience minimally.

Software-Defined Perimeter (SDP) is a zero-trust network-isolation defense model/framework which shows promise in addressing these areas. SDP dynamically configures one-to-one user-data connections as needed, reducing the number of resources which can be attacked per compromised user while maintaining access to resources for a majority of users [20]. An SDP is configured by assigning *roles*, which can represent account types, to users, which dictate acceptable user-behavior and accessible resources, represented as connections/edges in the network. In other words, these account types act as a gateway between network users and network resources [20]. Configuring an SDP can be considered an optimization problem because the goal when configuring an SDP is to minimize the number of roles needed to minimize the security risks of the network.

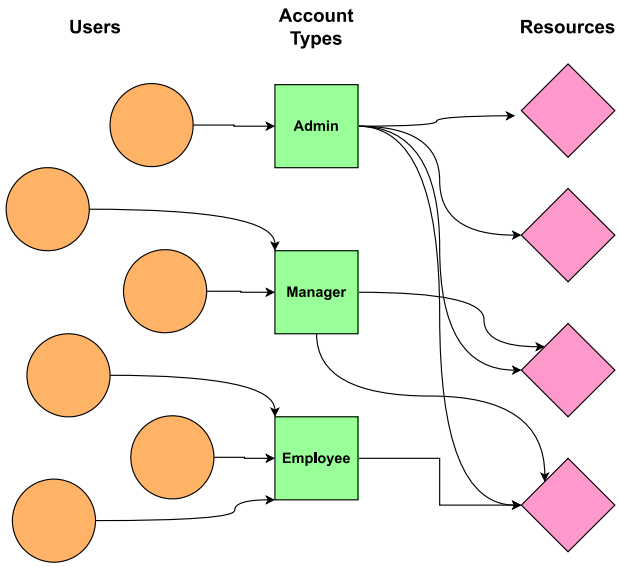

Figure 1: An example of an SDP configuration on a network where users are either employees, managers, or administration.

In this paper, we devise, implement, and experiment with a novel system called *SDPush* which can automatically configure and analyze SDP networks with user-specifications.[1] In doing so, we tackle the following research questions:

RQ1. Designing SDPs: How do we automatically design and optimize potential SDP networks?

[1]This work was submitted to the GECCO 2024 EC + DM Workshop Conference. The abridged version of this report can be found at the following DOI: 10.1145/3638530.3664155. This report is cited as follows in the GECCO 2024 version [8]

RQ2. Evaluation: How do we evaluate potential SDP networks? What are computationally cost-effective methods for doing so?

RQ3. Interpretation: How can we represent the SDP networks and choices made by the system in an interpretable manner?

RQ4. Limitations: What are the limitations of the our system in terms of the usefulness to an expert in their decision making process?

To address RQ1, we consider what must be given in order to define a unique network along with its desired characteristics. Furthermore, we consider what should be designed by the system (e.g. the number of account types, the count of users with an account type, permissions an account type grants, and more). We decide to use evolutionary computational methods for their ability to find solutions in large, non-systematic problem spaces and their previous success in automatically defining roles [30]. Thus, we design our system in the context of evolutionary computation, deliberating that the training set for our designer contains subsets of training cases aimed at evaluating different characteristics of an SDP network, i.e. a multi-objective system. Moreover, we assign evolutionary computation with the task of designing and evaluating roles to be used for a given network, where training cases provide analysis and explanations of the network, e.g. the security risks and accessibility of resources.

RQ2 presents a larger challenge as we must balance time complexity, accuracy, interpretability, and explainability. Ideally, we would simulate attack and defense strategies to evaluate security risks during the design process. However, we would prefer to provide an expert with suggestions in a reasonable time-frame, possibly 48 to 72 hours. Given this constraint, we must consider alternative methods of evaluation as simulations are computationally expensive. Thus, we derive a security risk estimation function based on previous work in developing SDP analysis frameworks [25]. We find this estimation function reduces the computational cost of the system enough to produce suggestions within our allotted time-frame. While the estimation function does not replicate simulations exactly, we find it is sufficient for guiding evolution towards useful results.

These discoveries lead us to dividing the responsibilities of our system into two phases. Phase one, which we call the *Designer*, designs a potential SDP configuration via evolutionary computation. The Designer is only concerned with designing feasible configurations to minimize security risks in a given network while adhering to provided user-specifications as best as possible. Thus, the results need not be interpretable or explainable to users. Phase two aims to transform the results of the Designer into a human-friendly format with explanations for design choices provided in some manner. Thus, we call this phase of the system the *Interpreter*.

Based on our answers to RQ2, RQ3 has various possibilities as the output of the Designer can be transformed into an acceptable input format for the Interpreter. In other words, by separating the system into two we can benefit from the ease of computation from one representation and the interpretability of another. For the Designer, we choose to represent the relationships between users, account types, and resources through bitstrings, so we can utilize bitstring genetic algorithms. Bitstrings have the advantage of being evaluated relatively faster than other representations but are not as interpretable to users [6, 7]. For the Interpreter, we choose to represent the Designer's given SDP configuration as an adjacency matrix of a graph. We then further evaluate the viability of the SDP configuration by using a co-evolutionary framework which uses Monte Carlo simulations to determine outcomes of attacker and defender scenarios on the network. We represent the decisions of the attackers and defenders as PUSH programs not only for interpretability as a programming language but also because genetic programming (GP) has shown to develop expressive and unique PUSH programs which can act as reasonable heuristics of attacker and defender behavior [3, 27, 28].

To give a comprehensive overview of our system, we discuss situations and parameters which SDPush cannot (yet) account for. In particular, SDPush requires an expert to provide a maximum number of account types, and if this argument is insufficient, then the run's suggestion will be uninformative. Also, while our system can account for several user-specifications for an SDP configuration, creating a subset of training cases and their respective error function is labor-intensive. These limitations provide avenues for future research.

In the next section, we will describe related work, including explicit use-cases of SDP and its strengths and weaknesses as well as explaining why evolutionary computation is a viable method for designing SDPs. We address RQ1, RQ2, and RQ3 in Section 3, Section 4, and Section 5 by devising our automatic SDP designing and explaining system SDPush and determining how we will represent networks and SDPs in our system. We then outline and conduct experiments in Sections 6 and 7. We conclude with a discussion of the results of our experiments, the limitations of our system, and future work which could be explored with SDPush.

2 RELATED WORK

An SDP can mitigate or completely defend against various cyber attacks such as server scanning, denial of service, password cracking, man-in-the-middle attacks, and many others once configured [5]. An example of an SDP is provided in Figure 1. In this SDP, the configuration has three account types: Admin, Manager, and Employee. Reasoning behind these configurations could be as follows. Since an admin must be able to secure the network at all times, they are given full access to the network. Note that only one admin user exists in this network. Since an admin may be necessary for this network and SDPs function under a zero-trust policy, this SDP configuration mitigates the potential security risks by reducing user count. A manager may require more resources in the network to complete their job, but they do not need the fine-grain control and complete control an admin does. Thus, the manager role connects to only two resources. Lastly, a company may have many employees that need to access resources in the network. In this simplified scenario, all employees need to access one resource. Thus, all employees are bucketed into the employee account type giving access to only one resource. Again since this a zero-trust network, we assume the more users we add to this account, the more likely this resource is to become compromised. However, we mitigate this security risk by assigning only one resource to be connected to this role. Therefore,

we have successfully configured this network to function with an SDP.

Despite SDP's proven effectiveness in various use-cases [10, 21], it is often not implemented as it requires an expert to manually design and analyze possible configurations for each unique network [20]. The amount of time an expert needs to design a configuration for a particular network drastically increases with network complexity and size as there is no systematic method for designing such [23].

Other systems/frameworks aimed at designing SDPs for given networks are implemented to consider only specialized environments such as Software Defined Networks, cloud-computing, visualization technologies, and the Internet of Things [10, 19, 21, 24]. Our work focuses on generalizing the designing process of SDPs for any environment that can be simplified to a set of resources and expected user-traffic. We also provide an automatic designing system while other works only provide systematic approaches. Furthermore, we provide analysis alongside configurations while other works only focus on providing either an SDP configuration or an analysis [10, 24, 25].

The use of evolutionary algorithms (EAs) for cybersecurity simulations is extensive. Simulations themselves have proven useful in modeling real-life applications/scencarios which other methods such as estimate functions fail to capture [18, 32] Garceia et al. provide a co-evolutionary framework called *RIVALS* which models attack/defense strategies as a simulator for real-life applications [9]. From this framework, co-evolutionary models have been used to simulate cyber attack patterns from publicly available datasets, identifying botnets in streaming data scenarios, and in modeling attacker and defender interactions in segmented networks [13, 15, 26]. Of particular relevance to this paper, EAs have been used in assisting experts in choosing defense strategies for a given network [14]. We draw on inspirations from these works for how we implement the Designer and Interpreter of SDPush. More specifically, we utilize the strategy of Hemberg et al. of using a simulation for initial analysis and following this with a co-evolutionary system which explores the adversarial spaces for segments of a network [13]. For the Interpreter, we apply the framework developed by Shlapentokh-Rothman et al. and utilize an adverserial co-evolutionary system where defenders (solution population) have their fitness determined by their ability to protect against attackers' (testing population); attackers, from their ability to overcome the defense strategies of the defenders [25].

Our work specifically utilizes a bitstring genetic algorithm (GA) and genetic programming (GP) to provide SDP networks and analysis respectively. Both evolutionary computational methods have been used extensively in the field of cybersecurity to solve large-scale complex problems or to provide analysis for existing systems. GAs have been used to design and evaluate networks at various scales [29, 31]. The aforementioned RIVALS framework is designed to accommodate various types of GAs in modeling cyber attacks such as malware intrusions in peer-to-peer (P2P) networks [9]. For the Designer, we choose to utilize a bitstring GA for its quick evaluation and Doerr et al.'s findings on faster genetic mutation operators [6, 7]. More generally, we choose to use a GA for designing SDPs as no systematic method for guided design and implementation of roles exists currently, and Suarez-Tangil et al. have shown that genetic programming can automatically produce rules for events based on event correlation [23, 30].

Genetic programming (GP) is an evolutionary method that solves specified computational problems via producing computer programs [16]. GP defines a problem's specifications via a set of training cases which, when used as a supervised learning technique, captures different aspects of the desired problem space. It evaluates evolved computer programs by measuring their ability to solve the desired problem. It achieves this by running each program on each training case in the provided training set and compares the programs' output(s) to the desired output(s), assigning an error value to each program. GP then uses these error values during parent selection to select programs from the evolved population to reproduce, and how many child programs each evolved program produces. Adversarial co-evolution functions similarly. However, the training set consists of another evolving population referred to as the testing population. The testing population is evaluated on its ability expose issues the solution population has not yet accounted for [22]. We choose to use GP in particular because it has proven useful in modeling cyber-attacks such as botnet detection or malware intrusion models for SDPs on P2P networks [15, 25].

The Interpreter of SDPush produces attacker and defender strategies in the form of PUSH programs. PUSH is a stack-based programming language designed for genetic and evolutionary algorithms where program instructions and data are stored on stacks [27, 28]. PUSH supports the evolution of expressive, modular architectures and complex control-flow structures achieved via self-manipulated code. It also supports cross-data manipulation and the ability to add new stacks to the language representing new data types. The reason PUSH is used in program synthesis is for its lack of errors. If an instruction wants to be executed, it must have all the necessary parameters on the data stacks to be executed. If an instruction fails to meet this criteria, then it does not attempt to perform the operation, or "no-ops". Thus, any potential erroneous code is ignored during interpretation of the program. This allows evolutionary practices to not be punished for exploring other domains of the problem space which may have previously been punished due to incomplete code.

We utilize this ability in the implementation of the Interpreter by providing a Network stack which allows defenders and attackers to directly interact with the network architecture provided by the Designer.

3 SDPUSH

We present a high-level overview of the SDPush system in Figure 2. This diagram outlines the control flow of the SDPush system along with the input/output relationships throughout the system and its two phases. In Sections 4 and 5, we will explore the inner mechanisms of the first and second phases which design and analyze SDP networks respectively. More specifically, we will describe the purpose of each phase, the representations of individuals to evolve, our choices in hyperparameters, and our reasoning for implementing each system as they are. In the next section, we discuss the threat model SDPush uses for both phases.

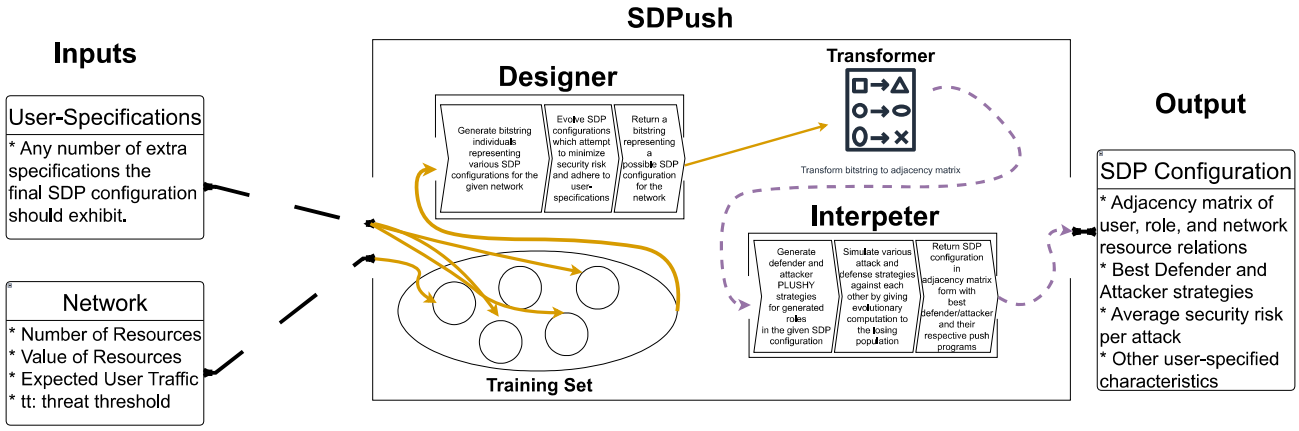

Figure 2: Diagram of the control flow of the SDPush system

3.1 Threat Model

We draw inspiration from Shlapentokh-Rothman et al.'s SDP analysis framework [25]. Let A, U, R represent the set of account types, users, and resources respectively. In our threat model, attackers and defenders aim to maximize and minimize compromised resources respectively. Defenders attempt to minimize the value of compromised resources by discovering compromised users $u_c \in U$ and removing them. Attackers try to maximize the value of compromised resources by exploiting undiscovered compromised users in the network. Attackers and defenders distribute a budget of attacking and defending strength respectively across the account types A. We use this budget as an abstraction of real-life resources dedicated to the tasks of defending or attacking. This distribution determines the probability they successfully complete their tasks relative to $a \in A$. Each attacker and defender can have a maximum budget of 10 they can distribute. Let $p_{at} = \frac{\texttt{attack\_alotted}}{10}, p_{de} = \frac{\texttt{defend\_alotted}}{10}$ be the probabilities of attackers and defenders succeeding respectively in their tasks for a user from a particular account type.

If a defender successfully removes all compromised users from a given account type so $|U_a| = 0$ where U_a is the set of users in account type a, then any resource R_a cannot be compromised via a. If $|U_a| > 0$, then attackers can attempt to exploit $|U_a|$ compromised users. If an attacker is successful in doing so, then all resources in R_a are compromised and thus $R_a \subseteq R_c$ where R_c is the set of compromised resources with their associated values. Otherwise, R_a remains uncompromised. Thus, we can quantify the objectives of the defenders and attackers in the threat model by the formula $\texttt{SecurityRisk} = \sum_{r \in R_c} v(r)$ where $v(r)$ is the value of resource r. The primary goal of the defenders/ attackers is to minimize/maximize this function's output.

4 THE DESIGNER

In this section, we address RQ1 by describing the first phase of the SDPush system called the Designer. This phase is responsible for designing an SDP configuration which minimizes security risk while also adhering to other constraints which the user specifies. We call these requirements the objectives of the Designer. We present a high-level overview of the Designer algorithm in Algorithm 1.

Algorithm 1 Designer

trainingset : cases evaluating other objectives are mixed with the main training set which evaluates security risk
tt : threat threshold of percentage of resource values that will be tolerated if lost
let B be the population of bitstrings representing SDP configurations
let $b \in B$ have the lowest security risk score of all individuals in B
$g \leftarrow 0$ ▷ g is the generation counter
while $g <$ max generations **do**
  evaluate all individuals in B on *trainingset*
  let b_g be the best bitstring in g
  if b_g has a lower security risk score than b **then**
    $b \leftarrow b_g$
  $B \leftarrow$ a population of children generated by B via parent selection and genetic operations
  $g \leftarrow g + 1$
return b

In the next section, we will explain how we represent a network and SDP architecture in bitstring form along with the benefits of such. In Section 4.2, we derive our security risk estimate function which this phase uses to model defender and attacker scenarios as a computationally cheaper method at the expense of accuracy. In Section 4.3, we explain how we are able to apply selection pressure during evolution to generate individuals that conform to user-specifications and other objectives while primarily minimizing security risk via lexicase selection.

4.1 Network and SDP Representation

The Designer phase of SDPush evaluates and manipulates bitstrings which represent an SDP configuration in relation to the network

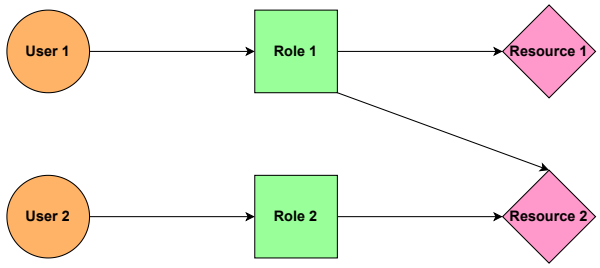

Figure 3: A visual representation of the SDP configuration "10011011" where $U = \{u_0, u_1\}, R = \{r_0, r_1\}$, and $A = \{a_0, a_1\}$.

specifications. By using a bitstring representation, the Designer can evaluate various types of objectives on a singular individual faster than with other representations such as the PUSH programming language which the Interpreter phase uses [7, 27].

Bitstrings encode SDP configurations relative to the account types defined in the network. The user provides an upper-limit to the number of account types the desired SDP may have, we call this A. If no upper-limit is provided, we assume $|A| = 20$. Each user and resource in the network gets $|A|$ bits within the bitstring representing their adjacency to account types in the network. Thus, the length of a bitstring individual b in the Designer is always $(|A| * |U|) + (|A| * |R|)$, where U is the set of network users and R is the set of network resources. We can acquire the sub-bitstring b_u of a particular user by slicing the original bitstring b as follows: $b_u = b[u_i : u_i + |A|]$, where $u_i \in U, 0 \leq i < |U|$ is a particular user's index. Similarly, we can acquire the sub-bitstring of a particular resource as follows: $b_r = b[|U| + r_j : |U| + r_j + |A|]$, where $r_j \in R, 0 \leq j < |R|$ is a particular resource's index. Note that the Designer assigns the first $|U| * |A|$ bits of b for user assignment and the rest for resource assignment.

By using these formulas, the Designer can keep configurations in bitstring format for evolvability and construct its corresponding SDP configuration for evaluation. Figure 3 provides an example of a network defined in bitstring form.

4.2 Security Risk Estimation

While simulations provide realistic feedback in the cases of modeling cyber attacks, evaluating candidate configurations in this manner is computationally expensive. Thus, we address RQ2 by examining how to best utilize a security risk estimate function to reduce computational costs. Let tt be the user-provided threat threshold representing the number of resources that is acceptable to have compromised in any given attack. If a generated SDP is estimated to lose less than or equal to tt resources, it is considered a potential SDP configuration for the network. We determine tt for a given SDP network with the Security Risk Score calculated below.

$$\texttt{SecurityRiskScore} = \left(\sum_{r \in R} \left(1 - \left(\prod_{a \in A_r} q_{at}^{q_{de} * |U_a|}\right)\right) * v(r)\right) / v(R)$$

The `SecurityRiskScore` estimation function iterates through each resource $r \in R$ calculating the probability that r is compromised and thus $\{r\} \subseteq R_c$. We calculate this by determining the probability the attacker compromises at least one user in U_a. The probability an attacker does not successfully compromise a user is q_{at}. The probability a defender does not successfully identify a non-compliant (vulnerable) user is q_{de}. The attacker has $(1 - p_{de}) * |U_a| = q_{de} * |U_a|$ chances to do so. The probability to compromise at least one user is multiplied by the value of the resource. We sum the estimated losses of the resources and return this as our security risk estimate. In the case that a user in the network cannot access at least one resource, we add a penalty to the security risk score given by the estimate function for each user not connected to a resource and vise versa.

To evaluate an SDP network using the estimate function, we randomly generate two arrays, D_d and D_a, of floats where the length of each array is $|A|$. The sum of each array is equal to 10.0. We assign array D_d to be the defender's budget distribution while array D_a is the attacker's budget distribution. For account $a_i \in A$ where $1 \leq i \leq |A|$, the probability the defender and attacker are not successful at their task is $1 - \frac{D_d[i]}{10}$ and $1 - \frac{D_a[i]}{10}$ respectively. We generate 100 of these array pairs to be used as standard inputs to the estimate error function for each SDP. Thus, every SDP network will be evaluated on their ability to minimize their security risks with these defender/attacker strategies.

4.3 Adhering to Other Objectives

While the Designer's primary concern is minimizing security risk, we want it to also consider given user-specifications during evolution. Examples of user-specifications include requiring one account type to connect to all resources in a network or allowing only 3 users to connect to any account type. We inject cases testing these specifications to varying levels of scrutiny into the main training set of the Designer with their own error functions. By using lexicase selection, we can filter candidate SDPs by their compliance to these objectives [12]. Lexicase selection gives the added benefit of letting different objective errors be evaluated at a different scale than the security risk errors, e.g. a boolean error value can function on a [0, 1] scale while security risk can function on a [0, 100] scale.

5 THE INTERPRETER

Here, we describe the second phase of the SDPush system called the Interpreter. This phase analyzes the SDP network generated from the previous phase by simulating defender/attacker scenarios. This phase is only concerned with analyzing the security risks, defender strategies, and attacker strategies as the user-specifications will have been met by the Designer. We present a high-level overview of the Interpreter algorithm in Algorithm 2.

In the following section, we explain the network and SDP representation during this phase of the SDPush system. In Section 5.2, we describe the simulation framework we use to analyze the potential SDP in further detail. In Section 5.3, we discuss the role of PUSH programs in representing defender and attacker strategies, along with the stacks and instructions used to do so.

Algorithm 2 Interpreter

```
attack: set of attackers where each attacker has their attack budget distribution and a PUSH program
defender: the current best defender against the attack set
s = 1: number of switches between defender and attacker evolution
max_s: maximum switches allowed
generate 10 attackers for initial defender training set
while s ≤ max_s do
    if s is odd then
        run defender GP to find defender_new
        if defender_new minimizes security risk better than defender then
            defender = defender_new
    else
        run attacker GP to find attacker_new
        attack = attack ⋃ {attacker_new}
    s = s + 1
return defender, attack
```

Algorithm 3 Bitstring Simplifier

```
let b be the given bitstring SDP
let A be the set of accounts in b
for a ∈ A do
    if a is not connected to any users OR a is not connected to any resources then
        remove a from all users and resources in b
    else
        if a = x ∈ A \ a then
            merge the users of both a and x into y = a = x
return b
```

5.1 Network and SDP Representation

The Interpreter phase of SDPush analyzes a given SDP network for security risks and presents its findings to the user. The Interpreter is only concerned with analyzing the security risks of the network under the provided SDP network. Thus, by using an adjacency matrix representation, we provide an interpretable format which can be quickly referenced when developing and understanding the defense/attack strategies. We present a high-level overview of the bitstring-to-matrix transformation algorithm in Algorithm 3.

Similar to the bitstrings the Designer uses, the adjacency matrix is constructed relative to the account types found in the network with users appearing first in the matrix. Thus, we utilize the formulas found in Section 4.1 to obtain the user and resource information and add this to our matrix in the order they appear in the bitstring.

5.2 SDP Simulation

We utilize the existing framework Shlapentokh-Rothman et al. developed for analyzing SDP networks' security risk [25]. In this framework, a Monte Carlo simulation is run with a pairing of a defender and attacker budget for the existing account types in the network. We use a simulation to evaluate the SDP during this phase as we are intently analyzing only one SDP network as compared to multiple. Therefore, we can take on the computational cost of simulating attacks on the network without taking an excessive amount of time to do so.

DESIGNER SPECIFICATIONS	
Population Size	1000
Max Generations	20000
Threat Threshold	0.50
Parent Selection	Lexicase Selection [12]
Variation	Bit-Flip & Uni. Crossover [7]
Mutation Rate	0.09
Initial Genome Size Range	$20 * (\|R\| + \|U\|)$
Max Account Types Created	20 **OR** user-provided

Table 1: The specifications for all runs of the Designer phase of SDPush. R is the set of resources in the given network and U is the set of expected users in the network.

Defenders are first given the opportunity to discover and remove compromised users in the network as described in Section 3.1. Attackers are then given an opportunity to exploit any remaining users in the network to compromise any resources the user can access via the account types they are attached to. The total value of the resources compromised by the exploited users is the security risk score for that simulation. We take the average of the security risk scores for a defender/attacker pairing and assign it as the result of the pairing.

5.3 Strategies as PUSH Programs

In the interest of generating defense and attack strategies that produce meaningful explanations, we do not directly evolved the budget distributions of the defenders and attackers as Shlapentokh-Rothman et al. does. We instead evolve PUSH programs which can use information regarding the network to generate the budget distributions across the account types. These PUSH programs take an adjacency matrix of a network as input and provide a vector of length $|A|$ as output.

We are able to generate these programs by using five stacks. The `exec`, `float`, `boolean`, and `vector` stacks and their respective instructions are implemented similarly to their Clojush implementations (explained in Section 6). The `network` stack is a novel stack type which allows PUSH programs to gather information about the provided network. The data stored on this stack are the users and resources in the network. We provide the standard stack manipulation instructions other stacks have in the PUSH language [27]. We also provide two instructions called `network_value` and `num_connections` which are polymorphic and take the top element from the `network` stack. If `network_value` is given a resource, it returns the value of the resource as a float. If it is given a user, it returns a float representing the total value of all resources the user can access via their account types. The `num_connections` instruction simply returns a float of resources a user has access to and vise versa.

INTERPRETER SPECIFICATIONS	
Defender Population Size	100
Attacker Population Size	100
Defender Max Generations	1000
Attacker Max Generations	1000
Defender Strength Range	[0, 10]
Attacker Strength Range	[0, 10]
Max Switches	20
Simulations per Pairing	100
Parent Selection	Epsilon-Lexicase Selection [17]
Variation	UMAD [11]
Mutation Rate	0.09 [11]
Initial Genome Size Range	[1, 200]

Table 2: The specifications for all runs of the Interpreter phase of SDPush.

6 METHODS

In this section, we discuss system-specific hyperparameters for the SDPush system and our method for implementing SDPush. We implement the SDPush system similarly to how Clojush[2], a PushGP implementation in the Lisp dialect Clojure, is implemented [27]. The code used in our experiments is available on GitHub[3]. A list of fixed specifications for the Designer and Interpreter phases of the SDPush system can be found in Table 1 and 2 respectively.

For our SDPush system, the user must specify the average number of expected users to use the network, the number and value of resources in the network, the threat threshold, tt, of the percentage of resource value that is acceptable to be at risk, and any other specifications the user desires. Once these arguments have been provided, the Designer's training set is constructed. As an example, assume a desired SDP network should have an account type connected to at least 60% of the resources in the network. The user provides, along with the network itself, this desired percentage and an error function which can be applied to bitstrings to determine if there exists at least one account type that follows this criteria. To provide more training data for the Designer to use, SDPush creates various training cases which checks if an SDP candidate has an account type that connects to 10%, 20%, ..., 60%, of the resources in the network.

7 EXPERIMENTS AND RESULTS

In this section, we exhibit our SDPush system's ability to automatically design, interpret, and explain configurations to a user for a given network which minimizes security risks and incorporates user-specifications. In the next section, we describe the runs performed to collect data about the system. In the sections that follow, we provide our analysis of different parts of the system. In Section 8, we provide our interpretation of the results.

[2]https://github.com/thelmuth/Clojush
[3]https://github.com/jgfrazie/SDPush

7.1 Setup

We devise a method for testing various inputs into the SDPush system in a systematic order. For every execution of the system, we provide a threat threshold of 50% and a maximum number of account types set at 20. We then input all possible combinations of network users and resources where the number of users and resources in the network could be [5, 10, 20, 40, 80, 160] for a total of 36 combinations. We run each of these combinations 150 times. The value of each resource in the network is randomly assigned an integer in the range [1, 50].

7.2 Security Risk Variance of Proposed SDPs

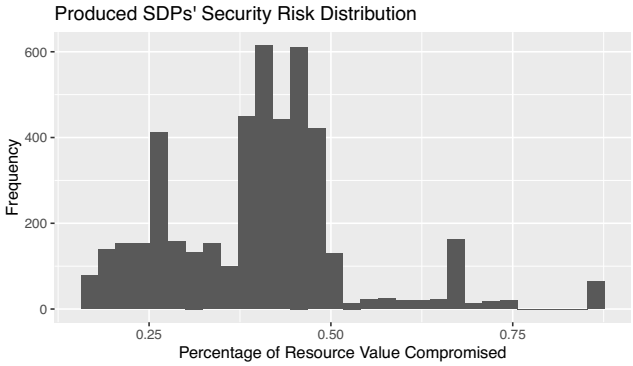

Figure 4: The simulated average percent total of resource value compromised in any attack against the best defense strategy of the automatically designed SDP

Figure 4 illustrates the distribution of security risks of generated SDP networks. To construct this distribution, we ran the Monte Carlo simulation from Section 5.2 on the outputted best defender against each attacker 100 times. The average percentage of the total resource values lost across the simulations was then used as the security risk for the outputted SDP. The large frequency observed at the 25% mark consists of a majority of runs where the number of users or the number of resources was less than or equal to the maximum number of account types.

7.3 Comparing the Security Risk Estimate to Simulations

Figure 5 provides a null distribution of the security risk evaluations of SDPs generated by SDPush. The purpose of a null distribution test is to analyze the equality of two sets of data. If the two sets of data being tested follow the null hypothesis (are equally comparable to each other), then there should be little observed difference between sample means. Otherwise, the two data sets are not comparable.

To construct this figure, we use the same method in Section 7.2 to obtain the simulator estimate for security risk. We then used the security risk estimate error function described in Section 4.2 on the outputted SDP as to obtain a security risk calculated via estimation. These two security risk scores act as our two data sets. We shuffle the labeling of our data between the two sets to obtain a new permutation of the two data sets. We then calculate the difference

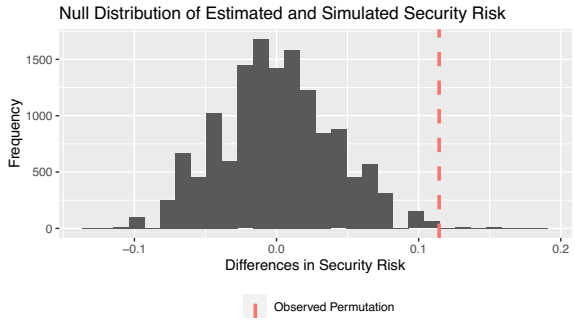

Figure 5: The null distribution of the security risk of designed SDPs using the security risk estimate function and the Monte Carlo simulator.

Algorithm 4 Best Defender for a 40-user 80-resource network

```
let strat be the budget distribution array
for a ∈ Accounts do
    let value be the distributed resource value
    for r ∈ Resources_a do
        value+ = v(r)/|Users_a|
    value = value/|Resources_a| strat[a] = value
return strat
```

in means of these sets and add it to the frequency plot. We repeat this process 10000 times. The vertical dotted line represents our original observed permutation relative to the shuffled permutations under the null hypothesis. Considering its position relative to the distribution, the null hypothesis should be rejected.

7.4 Effect of User-Specifications

For the experiments in this section, we provide the same threat threshold as the other experiments and set the maximum number of possible account types to 5. We define a network which contains 5 users and 5 resources each valued at 10, 20, 30, 40, and 50. Our reasoning behind this is to provide SDPush with an "easy" network to solve so we can observe the effect user-specifications have on the Designer.

We perform three executions of the system with their results found in Figure 6. The first execution has no user-specifications provided. The second execution has a user-specification to have at least one account type connected to at least 50% of the resources in the network. The third execution also has a user-specification, to have at least two account types connected to at least 50% of the resources in the network. The error function for these specifications will return the boolean `TRUE` if there exists account types of this requirement and `FALSE` otherwise.

7.5 Defender and Attacker Strategies

Several defender and attacker strategies were generated as "final" strategies outputted by SDPush. However, the most common and effective strategies followed the same high-level structure as Algorithm 4. In these strategies, the individual iterates through the accounts and checks the value of resources connected to that account. Using this information, some form of a defense value (usually a percent average of the total network value) was used as the budget for each account.

7.6 Networks Minimizing Security Risks Only

Here we provide example network structures configured by the Designer in Figures 7 and 8. While several strategies and designs were created by the Designer, most of them were a derivative of one of these two strategies.

8 DISCUSSION AND LIMITATIONS

In this section, we provide our interpretation of the results presented in Section 7. In Figure 4, we find the distribution suggests 1) the Designer's suggestions are fairly consistent and 2) the threat threshold provided by the user plays a critical role in determining which generated SDPs the Designer considers suitable for the network. However, this affect does not force the Designer to adhere strictly with this requirement if a better SDP can be configured as is evident by the values before the 50% mark. The large frequency before the 50% mark suggests the naive solution of connecting one user or resource to an account is effective in this scenario.

Figure 5 suggests our security risk estimate function and the Monte Carlo simulator cannot obtain similar results. Thus, the estimate function most likely is not a viable substitution for using a simulation framework. Furthermore, this suggests the Designer is not using the most accurate information when designing SDPs. Despite this flaw, the estimate error function is still able to guide evolution to producing valuable suggestions for users as evident by Figure 4. Therefore, in situations where computational cost is restricted, the use of security risk estimates is a viable substitute to simulator results. However, future work should address this lack of accuracy in the Designer while maintaining the reduced computational costs of the current implementation.

The SDP networks shown in Figure 6 suggest the user-specifications provided have an effect on the Designer when configuring SDPs. While the provided user-specifications configured the desired account-to-resource relationships, the specifications had an unintended effect on account-to-user relationships as well. This unintended effect could be detrimental to the quality of suggestions provided by SDPush as certain specifications could produce undesirable characteristics in the generated SDP network, causing the run of the system to be significantly less useful to the user. If we wanted to prevent this effect from happening, it would require the user to provide another user-specification discouraging the observed behavior. With the current implementation of the SDPush system, this process is laborious. Thus, future work should investigate ways of mitigating unintended side-effects of these user-specifications on produced SDP networks.

As for the quality of generated defender/attacker strategies, Algorithm 4 illustrates a relatively static selection of strategies the Interpreter chooses. In other words, SDPush is able to provide unique results on a case-by-case basis; however, the value and diversity of these results does not meet expectations. Most strategies being a derivative of Algorithm 4 could lead to three conclusions: 1) the Interpreter is unable to produce unique defender/attacker

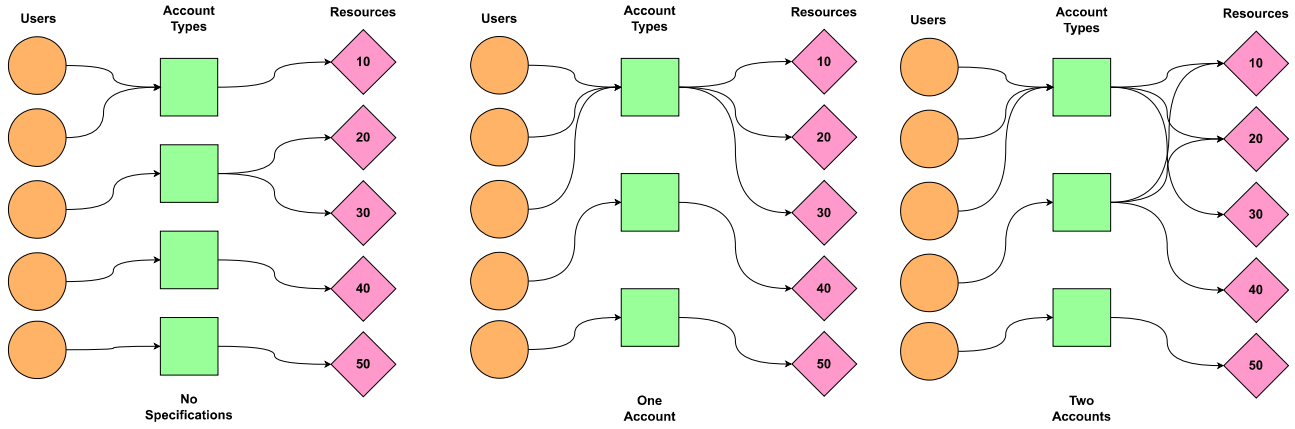

Figure 6: Different SDPs produced by SDPush. From left to right, no user-specifications were provided, one user-specification to have one account type connected to 50% of the resources, and one user-specification to have two account types connected to 50% of the resources.

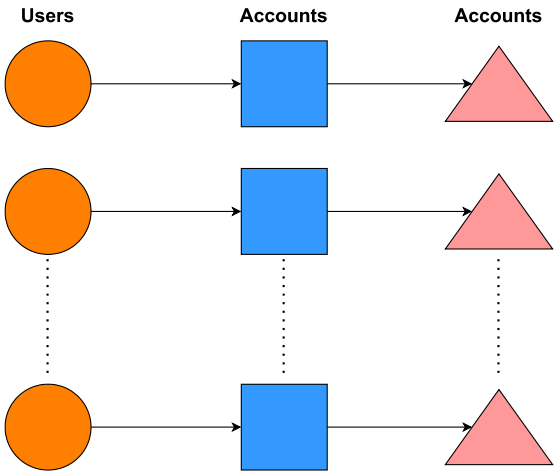

Figure 7: The most recurrent "simple" strategy the Designer used to construct minimal security risk SDP networks. It would pair users and resources in a one-to-one manner

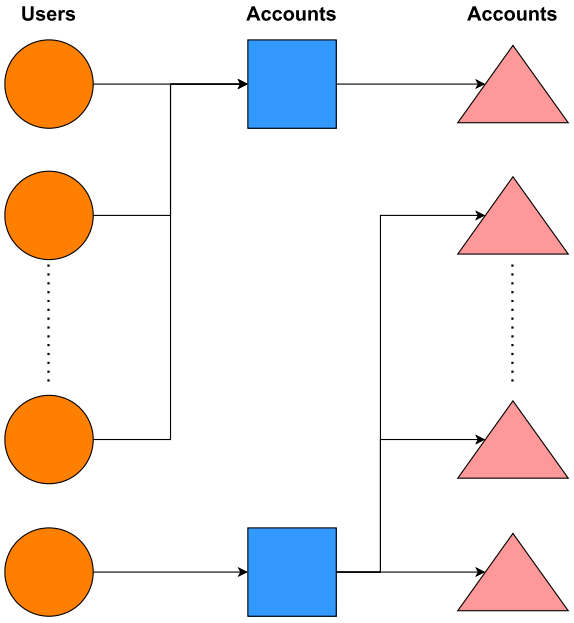

Figure 8: A recurring Designer strategy for configuring SDP networks. It would group all users into one account and all resources into another account attached to one resource and one user respectively.

strategies and requires improvement, 2) the Designer's choices limit the problem space of the Interpreter so much it contains the same set of minima in each run of the SDPush system, and/or 3) the nature of SDPs makes other possible strategies inferior to the evolved strategy provided.

Figure 7 and Figure 8 show SDPush's Designer phase as it is currently implemented is able to generate unique networks dependant on the inputted situation. Yet, most designs falling into a form of one of these strategies highlights the importance of user-specifications in the Designer's process. Not only do they give flexibility to the system and greater control to the user to help define their problem space more accurately, it also encourages a more diverse set of SDP configurations for networks as evident by Figure 6.

9 CONCLUSIONS AND FUTURE WORK

Software Defined Perimeter (SDP) stands as a strong contender for the future of network defense as it provides enhanced security

measures for relatively minimal cost. However, SDP requires an expert to manually design and analyze possible configurations for each unique network with no systematic guide for doing so.

We introduced our novel system SDPush: a two phase evolutionary computational system which can automatically design and analyze SDP networks. Our experiments demonstrate how SDPush can design fairly consistent configurations for networks which can take into consideration other specifications which do not directly involve minimizing security risks.

Although we have illustrated SDPush's ability to design and analyze SDP networks, there are avenues available for improvement. Future work should focus on incorporating more human interaction during the Designer phase and iterating on the security risk estimate function in order to produce more accurate and suitable networks more consistently. We also believe it is possible the Designer could benefit from the results of the Interpreter in iterating on a design. Furthermore, we anticipate a more user-friendly format for presenting the analysis of the network to arise other than just presenting the best defenders and attackers to the user.

As for quality of produced suggestions and explanations, we believe future work should attempt to coerce evolution to find more diverse solutions throughout the entirety of the SDPush system without requiring the use of user-specifications to do so. We also expect different methods of representing and evolving networks to be found which can improve many of the aspects discussed above.

ACKNOWLEDGMENTS

We would like to thank Sally Cockburn, Sarah Morrison-Smith, Jack Feser, and Lee Spector for discussions which helped develop this work.

REFERENCES

[1] [n. d.]. Cost of a data breach in the U.S. 2023. https://www.ibm.com/downloads/cas/E3G5JMBP. Accessed: 2024-2-4.

[2] [n. d.]. Number of data breaches and victims U.S. 2023. https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/. Accessed: 2024-4-5.

[3] Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. 2009. *Exploring Hyper-heuristic Methodologies with Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 177–201. https://doi.org/10.1007/978-3-642-01799-5_6

[4] Long Cheng, Fang Liu, and Danfeng Daphne Yao. 2017. Enterprise data breach: causes, challenges, prevention, and future directions: Enterprise data breach. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7 (06 2017), e1211. https://doi.org/10.1002/widm.1211

[5] Software Defined Perimeter Working Group-Cloud Security Alliance (CSA). 2013. Software Defined Perimeter. http://downloads.cloudsecurityalliance.org/initiatives/sdp/SoftwareDefinedPerimeter.pdf. (12 2013).

[6] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. 2017. Fast genetic algorithms. In *Proceedings of the genetic and evolutionary computation conference*. 777–784.

[7] Agoston E. Eiben and J. E. Smith. 2016. *Introduction to evolutionary computing*. Springer.

[8] James Frazier. 2024. *Applications of Artificial Intelligence to Information Privacy*. Senior Fellowship. Computer Science, Hamilton College. https://digitalcommons.hamilton.edu/cpsi_theses/1/

[9] Dennis Garcia, Anthony Erb Lugo, Erik Hemberg, and Una-May O'Reilly. 2017. Investigating coevolutionary archive based genetic algorithms on cyber defense networks. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1455–1462. https://doi.org/10.1145/3067695.3076081

[10] K Griffith. 2018. Software-Defined Perimeter Remains Undefeated in Hackathon.

[11] Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1127–1134. https://doi.org/10.1145/3205455.3205603

[12] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. https://doi.org/doi:10.1109/TEVC.2014.2362729

[13] Erik Hemberg, Joseph R. Zipkin, Richard W. Skowyra, Neal Wagner, and Una-May O'Reilly. 2018. Adversarial co-evolution of attack and defense in a segmented computer network environment. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1648–1655. https://doi.org/10.1145/3205651.3208287

[14] Hui Jin, Senlei Zhang, Bin Zhang, Shuqin Dong, Xiaohu Liu, Hengwei Zhang, and Jinglei Tan. 2023. Evolutionary game decision-making method for network attack and defense based on regret minimization algorithm. *Journal of King Saud University - Computer and Information Sciences* 35, 3 (2023), 292–302. https://doi.org/10.1016/j.jksuci.2023.01.018

[15] Sara Khanchi, Ali Vahdat, Malcolm I. Heywood, and A. Nur Zincir-Heywood. 2018. On botnet detection with genetic programming under streaming data label budgets and class imbalance. *Swarm and Evolutionary Computation* 39 (2018), 123–140. https://doi.org/10.1016/j.swevo.2017.09.008

[16] John R Koza. 1992. Genetic programming: on the programming of computers by means of natural selection Cambridge. *MA: MIT Press.[Google Scholar]* (1992).

[17] William La Cava, Thomas Helmuth, Lee Spector, and Jason H. Moore. 2019. A Probabilistic and Multi-Objective Analysis of Lexicase Selection and ϵ-Lexicase Selection. *Evolutionary Computation* 27, 3 (09 2019), 377–402. https://doi.org/10.1162/evco_a_00224 arXiv:https://direct.mit.edu/evco/article-pdf/27/3/377/1858632/evco_a_00224.pdf

[18] Mona Lange, Alexander Kott, Noam Ben-Asher, Wim Mees, Nazife Baykal, Cristian-Mihai Vidu, Matteo Merialdo, Marek Malowidzki, and Bhopinder Madahar. 2017. Recommendations for Model-Driven Paradigms for Integrated Approaches to Cyber Defense. arXiv:1703.03306 [cs.CR]

[19] Michael Lefebvre, Suku Nair, Daniel W. Engels, and Dwight Horne. 2021. Building a Software Defined Perimeter (SDP) for Network Introspection. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 91–95. https://doi.org/10.1109/NFV-SDN53031.2021.9665152

[20] Abdallah Moubayed, Ahmed Refaey, and Abdallah Shami. 2019. Software-Defined Perimeter (SDP): State of the Art Secure Solution for Modern Networks. *IEEE Network* 33, 5 (2019), 226–233. https://doi.org/10.1109/MNET.2019.1800324

[21] Yangchen Palmo, Shigeaki Tanimoto, Hiroyuki Sato, and Atsushi Kanai. 2023. Optimal Federation Method for Embedding Internet of Things in Software-Defined Perimeter. *IEEE Consumer Electronics Magazine* 12, 5 (2023), 68–75. https://doi.org/10.1109/MCE.2022.3207862

[22] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. De Jong. 2012. *Coevolutionary Principles*. Springer Berlin Heidelberg, Berlin, Heidelberg, 987–1033. https://doi.org/10.1007/978-3-540-92910-9_31

[23] Igor Saenko and Igor Kotenko. 2018. Genetic algorithms for role mining in critical infrastructure data spaces. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1688–1695. https://doi.org/10.1145/3205651.3208283

[24] Ahmed Sallam, Ahmed Refaey, and Abdallah Shami. 2019. On the Security of SDN: A Completed Secure and Scalable Framework Using the Software-Defined Perimeter. *IEEE Access* 7 (2019), 146577–146587. https://doi.org/10.1109/ACCESS.2019.2939780

[25] Michal Shlapentokh-Rothman, Erik Hemberg, and Una-May O'Reilly. 2020. Securing the software defined perimeter with evolutionary co-optimization. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1528–1536. https://doi.org/10.1145/3377929.3398085

[26] Michal Shlapentokh-Rothman, Jonathan Kelly, Avital Baral, Erik Hemberg, and Una-May O'Reilly. 2021. Coevolutionary modeling of cyber attack patterns and mitigations using public datasets. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 714–722. https://doi.org/10.1145/3449639.3459351

[27] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation* (Washington DC, USA) *(GECCO '05)*. Association for Computing Machinery, New York, NY, USA, 1689–1696. https://doi.org/10.1145/1068009.1068292

[28] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3 (03 2002), 7–40. https://doi.org/10.1023/A:1014538503543

[29] LV Stepanov, AS Koltsov, and AV Parinov. 2021. Evaluating the cybersecurity of an enterprise based on a genetic algorithm. In *Advances in Automation II: Proceedings of the International Russian Automation Conference, RusAutoConf2020, September 6-12, 2020, Sochi, Russia*. Springer, 580–590.

[30] Guillermo Suarez-Tangil, Esther Palomar, José María De Fuentes, Jorge Blasco, and Arturo Ribagorda. 2009. Automatic Rule Generation Based on Genetic Programming for Event Correlation. *Advances in Intelligent and Soft Computing* 63, 127–134. https://doi.org/10.1007/978-3-642-04091-7_16

[31] P Subashini, M Krishnaveni, TT Dhivyaprabha, and R Shanmugavalli. 2020. Review on intelligent algorithms for cyber security. In *Handbook of Research on Machine and Deep Learning Applications for Cyber Security*. IGI Global, 1–22.

[32] Allan Wollaber, Jaime Peñna, Benjamin Blease, Leslie Shing, Kenneth Alperin, Serge Vilvovsky, Pierre Trepagnier, Neal Wagner, and Leslie Leonard. 2019. Proactive Cyber Situation Awareness via High Performance Computing. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916528