Improving evolution of service configurations for moving target defense

Ernesto Serrano-Collado
University of Granada
Granada, Spain
info@ernesto.es

Mario García-Valdez
Instituto Tecnológico de Tijuana
mario@tectijuana.edu.mx

Juan J. Merelo-Guervós
Dept. of Computer Architecture and Technology
University of Granada
Granada, Spain
jmerelo@ugr.es

***Abstract*—The term *moving target defense* or MTD describes a series of techniques that change the configuration of an Internet-facing system; in general, the technique consists of changing the visible configuration to avoid offering a fixed target to service profiling techniques. Additionally, configurations need to be as secure as possible and, since change needs to be frequent, to generate also as many as possible. We previously introduced a proof of concept where we used a simplified evolutionary algorithm for generating these configurations. In this paper we improve this algorithm, trying to adapt it to the specific characteristics of the fitness landscape, and also looking at finding as many solutions as possible.**

***Index Terms*—Security, cyberattacks, performance evaluation.**

I. Introduction

While security is a constant concern in modern computer systems, the amount of services a typical application relies on and offers, and the sheer quantity of services and microservices, modern cloud-native, applications are composed of, makes extremely complicated to create configurations, for every one of them, that are at the same time secure and performant.

The wide variety of attacks and attack techniques also makes it difficult to create a single, static defense that can deflect every possible attack an interested third party might mount. Defense needs then to adapt to be able to confuse, deflect or avoid this kind of attacks. As an example, we can simply imagine that the name or IP of a node in a service is constantly changing; the attacker will be unable to use stored information (such as vulnerabilities) from that particular node to, later on, scale privileges, extract information (exfiltrate) from the net, or simply check if that service is up. Fortunately, modern cloud-native deployments facilitate that kind of defense: since the whole deployment is software defined, we can embed these changes within the deployment instructions themselves.

The kind of defense technique that tries to present a variable target to possible attackers is called *moving target defense* or MTD. The concept was proposed initially by the Federal Networking and Information Technology Research and Development (NITRD) Program for the first time in 2009 [1], and presented in a series of documents [2] and books that bind the papers presented in the first symposium dedicated to the topic [3]. The moving target defense [4]–[6] does not specify either the kind of attack that defense is being put up against, which could be from privilege escalation to denial of service attacks, the service that is being hardened or secured using this technique, which can go from a web or proxy server to a software defined network [7], or the kind of technique that is used to generate a moving target, which can also be simple randomization [8] of the user-facing information through churn, that is, changing often from a set of pre-established configurations through more elaborate systems like evolutionary algorithms [9] that, at the same time, optimize security or some other measure, like performance.

Our previous paper [10] was a proof of concept and tested the framework we have created for evolving a set of configurations that can be used in a MTD policy. Our target was hardening `nginx` installations and we used as a fitness function *Zed Attack Proxy* (ZAP), an open source tool that gives as a score for an installation the number of *alerts*, or possible security vulnerabilities, it raises. We tested different configurations and found that evolutionary algorithms are able to generate configurations with a low score (lower is better), and also that every execution of the algorithm yields several configurations with the same fitness, which can then be used straight away to change the configuration of the server.

However, that was intended as an initial exploration of the concept of using evolutionary algorithms to generate low-vulnerability and diverse nginx configurations. We needed to explore the possibilities of the evolutionary algorithm further, by tuning its parameters so that better configurations can be found with less evaluations. Also, we needed to explore different possibilities of the scoring tool, to check which mode would be better for the MTD task. These will be the two main objectives of this paper.

The rest of the paper is organized as follows: next we will present the state of the art in evolutionary methods applied to MTD; the next Section III will present the methodology used in this paper, followed by the experimental results, finishing with our conclusions (in Section IV and future lines of work.

II. State of the art

MTD was proposed by the first time in 2009 [1] by an organism called NITRD as part of an officially sponsored research program to improve the cyberdefense skills in the

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

United States. MTD is targeted towards making what is called the attack surface [11], that is, the different mechanisms by which the attacker would be able to gain access, unpredictable [3], and thus rendering attacks against it either too expensive or too complex to pursue, possibly forcing the attacker to choose some other, more affordable, place. For instance, an attacker analyzing byte patterns coming from different nodes such as the one described in [12] will find those patterns disrupted, and so profiling of specific nodes impossible.

This program was pursued using different kind of techniques, of which a good initial survey was made in [4], reexamined in [13] and more recently in [5], [6], [14]. MTD is used as a defense as well as detection technique [15], [16]; for instance, it can be used to deflect distributed denial of service attacks [17]; besides, it has been proved effective against exfiltration techniques via the use of an open source framework called MoonRaker [18] or to protect software defined networks [19]. Several techniques have been applied recently; for instance, natural randomization in services can be enhanced [20]; or, beyond the technique that is used, deep reinforcement learning can try and find the best moment for changing configurations [21], a topic that is normally left behind. These techniques have been surveyed in [14], [22], to which we direct the interested reader.

However, in this paper we focus on those that use evolutionary algorithms as a method of optimization as well as generation of new configurations; evolutionary algorithms are no strangers in the cybersecurity world, and in fact, since early on, they were applied to intrusion detection systems [23]. It was only natural that they were also applied, since the inception of the technique, to MTD. An evolutionary-like bioinspired algorithm called *symbiotic embedded machines* (SEM) was proposed by Cui and Stolfo [24] as a methodology for *injecting* code into systems that would behave in a way that would be similar to a symbiotically-induced immune system. Besides that principled biological inspiration, SEMs used mutation as a mechanism for avoiding signature based detection methods and thus become a MTD system.

Other early MTD solutions included the use of rotating virtual webservers [25], every one with a different attack surface, to avoid predictability and achieve the variable attack surface that was being sought. However, while this was a practical and actionable kind of defense, no specific technique was proposed to individually configure every virtual server, proposing instead manual configuration of web servers (such as nginx and Apache), combined with plug-ins (some of which do not actually work together). A similar technique, taken to the cloud, was proposed by Peng et al. [26]: a specific mechanism that uses different cloud instances and procedures for moving virtual machines between them; still, no other mechanism was proposed to establish these configurations, which were simply left to being designed by hand, as long as there were enough of them.

Bioinspired solutions filled that gap: after the early *bioinspired* approaches to MTD, explicit methodologies that used evolutionary algorithms were conceptually described for the first time by Crouse and Fulp in [27]. This was intended mainly as a proof of concept, and describes 80 parameters, of which just half are evolved. The GA minimizes the number of vulnerabilities, but the study also emphasizes the degree of diversity achieved by successive generations in the GA, which impact on the diversity needed by the MTD. Lucas et al. in [28] applied those theoretical concepts to a framework called EAMT, a Python-based system that uses evolutionary algorithms to create new configurations, which are then implemented in a virtual machine and scored using scanning tools such as Nessus. Later on, John et al. [9] make a more explicit and practical use of an evolutionary algorithm, describing a host-level defense system, that is, one that operates at the level of a single node in the network, not network-wide, and works on the configuration of the Apache server, evolving them and evaluating at the parameter level using the so called CVSS score [29], a score that is computed from a the categories of vulnerabilities detected and how severe they are. These two systems highlighted the need for, first, a practical way of applying the MTD to an actual system, to the point of implementing it in a real virtual machine, and second, the problematic of scoring the generated configurations. In the next section we will explain our proposed solutions to these two problems.

III. Methodology, experimental setup and results

As in our previous paper [10], we have chosen nginx; it's a very popular static web server, which is also used as an inverse proxy, API gateway and load balancer. Latest versions ofnginx (1.17.x) have more than 700 configuration directives. As a matter of fact, nginx has been the target of optimization by evolutionary algorithms recently [30], but this is not the main focus of our work presently.

These directives affect in different ways the behavior of the web site (or service) that is behind it, or simply change the values in the response headers; we will show the ones we will working with next. The following Subsection III-B will outline the setup actually used for running the experiments, and results will be presented last in Subsection III-C.

A. Description of the attack surface parameters

From the more than 700 directives available in nginx we chose nine (shown in Table III-A), the same used in a previous paper in the same line of work [10]. Also we get from the same paper 6 HTTP headers (shown in Table II), a set of directives that affect how the browser processes the received pages. They add up a subset of fifteen directives, that also matches a few of the more extended DISA STIG recommendations for hardening webservers based in the CVSS score [31]. Most of these values are defined in the original document as Apache HTTP server configuration values but we look for the nginx equivalent. The nginx directives that have been used here paper (as well as in [10]) and their equivalent STIG ID are shown in Table III-A along with the range of values for generation and ulterior evolution. They are also described next:

STIG ID	Directive name	Possible values
V-13730	worker_connections	512 - 2048
V-13726	keepalive_timeout	10 - 120
V-13732	disable_symlinks	True/False
V-13735	autoindex	True/False
V-13724	send_timeout	True/False
V-13738	large_client_header_buffers	512 - 2048
V-13736	client_max_body_size	512 - 2048
V-6724	server_tokens	True/False
	gzip	True/False

TABLE I
LIST OF nginx DIRECTIVES WHOSE VALUE IS EVOLVED IN THIS WORK

- `worker_connections`: Number of concurrent connections opened per process. In general, this will be neutral with respect to security, but it will help create a variable attack surface; it will also be related to performance, although at this stage we are not evaluating it.
- `keepalive_timeout`: Time to wait for new connections before closing the current one.
- `send_timeout`: Defines the maximum allowed time to transmit a response to the client. Sixty seconds by default. As in the previous case, these values have no influence on the security of the web site, but they do change the timing and content of responses.
- `disable_symlinks`: Allows returning symbolic links as files. When switched off (default value) accessing a file that is a symbolic link raises a denied access error. Although, in this case, this could be a security problem, it might not be so if there are no actual symbolic links in the site, in which case, it could be used to escalate privileges anyway.
- `autoindex`: Allows the generation of a page listing the content of the current directory. Set to off by default. This is also a security concern, because it might reveal information about hidden files not linked elsewhere.
- `large_client_header_buffers`: Number and size of buffers for large client requests headers. We are just evolving the size of the buffers, leaving a default number of four of them.
- `client_max_body_size`: The maximum size allowed per a client request body. If the client exceed this value the server will return an error. Again, there's no direct security implications for this directive.
- `server_tokens`: Return some server info in the `Server` response header. By default it shows the nginx version. The main implication of this is the revelation of information about the server, which is shown, for instance, in 404 pages. Even if it's on, fake information can (and, in fact, will) be generated by the `Server` directive we will use below, so it's not so much security-related as variable-attack-surface related.
- `gzip`: Enables the compression of HTTP responses. As in most cases above, it's not a security relative directive but adds some entropy to the generated configurations.

nginx sends some HTTP headers, whose value can be configured via the configuration file. These are presented next, with possible values represented in Table II.

TABLE II
SELECTED LIST OF DIRECTIVES AFFECTING HTTP HEADERS, AND THE VALUES THAT WE ARE USING IN THIS PAPER.

Header name	Possible values
X-Frame-Options	SAMEORIGIN ALLOW-FROM DENY WRONG VALUE
X-Powered-By	PHP/5.3.3 PHP/5.6.8 PHP/7.2.1 Django2.2 nginx/1.16.0
X-Content-Type-Options	nosniff
Server	apache caddy nginx/1.16.0
X-XSS-Protection	0 1 1; mode=block
Content-Security-Policy	default-src 'self' default-src 'none' default-src 'host *.google.com'

- `X-Frame-Options`: Tells the browser to don't allow embedding the page in HTML frames. This is useful to prevent *clickjacking* attacks where the attackers set a malicious transparent overlay layer on top of a real page.
- `X-Powered-By`: This header has a similar behavior as the `server_tokens` directive. Show the name and version of the application that generated the response. Setting different values does not affect directly the security by itself but adds entropy to the generated configurations.
- `X-Content-Type-Options`: Tells browser the requested document *MIME*. This is useful to avoid 'MIME type sniffing' attacks where the attacker changes the requested document to do a cross-site scripting attack.
- `server`: This directive is related to the `Server` HTTP header, which is used to communicate to browsers metadata about the server software used by the application. This can be as informative or as misleading as we want; as a matter of fact, it is a good practice not to give too extensive information of software versions, but we can cheat the attacker telling wrong server version info. Doesn't affect directly to the security but adds entropy to the generated configurations. This directive, along with `X-Powered-By`, are mainly used for informative or statistics purposes and do not really change anything either in content or how it is rendered by the server.
- `X-XSS-Protection`: This response header was a built-in filter firstly released by Microsoft for Internet Explorer and later by Google for Chrome that stopped pages from loading when they detect cross-site scripting (XSS) attacks. Although this header is unnecessary in today's browsers adds entropy to the generated configurations.
- `Content-Security-Policy`: This header tells the browser to avoid loading of some kind of content. we can set this directive to different values to avoid the

load of certain kinds of content. We can allow content only if it's loaded from the same page (`self`), loading from nowhere (`none`), or including a specific domain `hosts:*.example.com`.

The problem of optimizing security in the configuration is two-fold: some of these directives are security-neutral and do not actually alter the security score, they simply add to overall byte-level diversity; some of them, by themselves, will make a site more secure; in some other cases, it will be its combination what makes it safer. That is why a global optimization algorithm is needed to get variable, and also secure, attack surface.

B. Experimental setup

The most important part of the evolutionary algorithm is designing correctly the fitness function that is going to be tested and used. A configuration is meaningless without content behind, and we need to choose what is going to be the content. In our previous paper [10], a basic application was created, and additionally, a intentionally vulnerable application, OWASP's Juice Shop [32], were also tested. Since their vulnerabilities are different, the scores are going to be different. In this paper, we will focus on the Juice Shop, which is intended to be vulnerable and is thus a bigger challenge for the evolutionary algorithm.

The setup used for computing this score is exactly the same as in the last paper: standard docker containers for the juice shop and the ZAP API were composed (using Docker Compose) together with the container hosting the evolutionary algorithm, which calls from the fitness function the ZAP library, written in Python. What ZAP does is to run over all pages in the site, making different attacks and raising alerts if they detect any vulnerability; the score is equivalent to the number of alerts raised. From these alerts a CVSS standard score could be computed, but this is a direct transformation so we will stick to this score for our evolutionary algorithm. A 0 score will imply a totally secure configuration, while higher scores are related to the number of alerts. Please bear in mind that it is not a linear scale: a score of 58 does not indicate that the site is 20 times less secure than one with score equal to 3, it can mean that a new alert has been raised on 55 different pages. As a matter of fact, a score equal to 3 is due to a single alert raised three times in 3 different web pages of the Juice Shop store. This also implies that a score of 58 does not really indicate a very insecure place; it just means that it is less secure than a configuration with score equal to 3. Additionally, we are using a score of 999 as indicative of a invalid configuration, as checked by nginx command line.

The evaluation to obtain the fitness of every configuration is then performed in the following sequence:

- The population stores a representation of the configuration as a Python vector. That vector is processed through the `nginx-config-builder` library to generate a configuration file, that takes a random name to avoid race conditions.
- We use the `nginx -c` command line to check this file for correctness. Incorrect configurations should eventually be eliminated from the evolutionary process, so we assign them a cost of 999.
- We start the nginx server; since it takes a certain amount of time to tear down the previous instance, we make sure before that there is no server running; this measure avoids clashing on the occupation of a port but, more importantly, we take care to check that effectively the configuration we are checking is the one that has been generated by this specific chromosome.
- Once the site is back online, it can be scanned by the ZAP proxy, which is launched from a container. ZAP examines the whole site, making requests for every URL in it, and examining the response, including headers. From this, it generates a list of alerts that we simply count to compute the fitness (or, in this case, cost, since it's going to be minimized).
- The server is killed, with several checks until the process has been effectively eliminated.

TABLE III
DISTRIBUTION OF ZAP PASSIVE SCANNING SCORES

Zap Score	Number of instances
3	2
33	262
34	160
35	4312
999	2943

Although startup and teardown need some time to make sure that they have effectively happened, The more time-consuming part of this process is the scanning, which might take up to 15 seconds; this means that evaluation is a very slow process, which is a problem. Besides, ZAP is able to perform two types of scanning: passive and active. In our previous paper, we focused on active scanning, which makes requests and examines responses; however, one of the things we wanted to try in this case was passive scanning, which besides checks headers and performs other analyses on the requests and responses. Before actually using it in the evolutionary algorithm, we generated a series of random configurations and tested its score; its distribution is shown in Table III. There are several problems with this fitness landscape: there are just a few values, which makes evolution extremely difficult, since most of the fitness landscape is flat, and most changes in configuration will not result in a change of fitness. But the bigger problem is the amount of time passive scanning adds, which makes this way of evaluating fitness score totally impracticable. Eventually, we will have to use only active scanning, the same as in the initial paper.

As in our previous paper, [10], we have designed the rest of the algorithm in Python, to be able to accommodate the ZAPpy library which is written in that language; Python has a reasonable performance on evolutionary algorithms [33], but in this case the bottleneck is actually scanning, with evolutionary operations themselves taking but a very small fraction of the

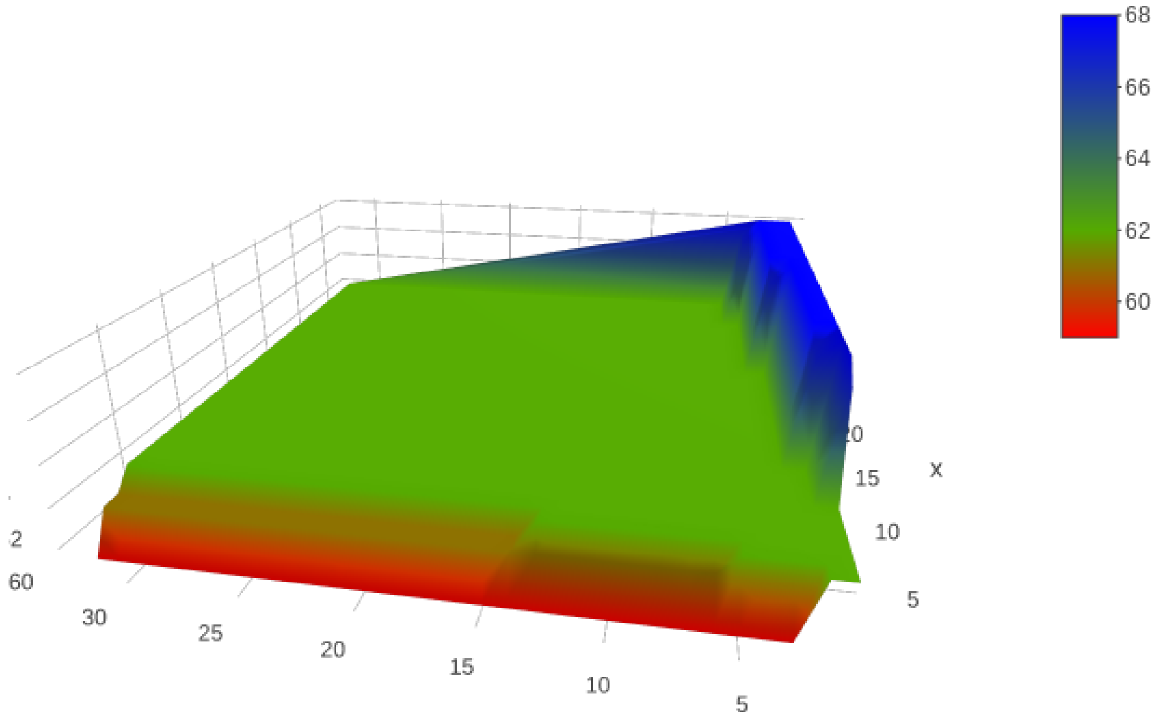

Fig. 1. 3D mesh representation of evolution of fitness, with z axis representing score (lower is better), y axis representing simply the rank order of every individual in the population, and x axis representing generations; evolution proceeds from right to left.

total time, so this choice is not really critical. But one of the main problems of our former implementation was that there was very little exploration of the search space, with final values being mostly small modifications of configurations that were already present in the initial population. There was little actual improvement of score during evolution, which was probably due to a naive implementation of the evolutionary algorithm, so the main objective of this research was to improve the evolutionary algorithm so that exploration is better and is able to find solutions way beyond the immediate vicinity of the initial set of solutions. Of course, this might also be due to the fact that we were only doing 15 generations, which is why we are increasing this number during the current phase of research.

The next part of the implementation of the evolutionary algorithms is choosing a data structure that is able to hold it efficiently and expressively. We have used a simple Python list of 15 integers, one per directive; this list is converted to a configuration file using a specific Python library, `nginx.config`, which makes easier the generation of configuration files. When the chromosome is evaluated, it becomes a two-element list, whose first element is the fitness and a second containing the data structure itself. This speeds sorting of the population, since these can simply be sorted by the first element in the list.

In [10] we made a proof of concept, that resulted in the genetic operators having more or less the same influence in the final result. However, the mutation operator used was too explorative, since it changed a value in a gene by a random one. This contributed to a high diversity, which is important and something we are interested in, but in the other hand we also need a mutation operator with a certain degree of exploitation, so in this paper we have used an arithmetic mutation that changes a gene value by ± 1, circling back or forward to the value extremes if they are reached. This *circular* arithmetic operator ensures that the values of every chromosome are kept within its allowed range, and thus, besides performing exploration of the space in a smoother way, eliminates invalid individuals from the population in very few generations.

The other variation operator, crossover, will be 2-point crossover, and it will return a single value, with pieces taken from both parents. We have chosen only this one, as opposed to the 1-point crossover used in [10].

While in the previous paper we were using a simple rank-based, non-fitness-proportional selection, which probably resulted in less exploitation of the best results, in this paper we have changed that to 2-tournament selection, which increases the selective pressure, eliminating invalid individuals more efficiently.

The source for the evolutionary algorithm is open and hosted in the same repository as this paper, with a free license. It follows a standard evolutionary algorithm, with selection via 2-individual tournament, crossover of two randomly picked individuals, followed by mutation. The old and new population are ranked, and just the best individuals are included in the following generation. In this case, we were more focused on how to analyze the fitness of every individual than on the evolutionary algorithm itself, which, in any case, is not facing a hard problem.

This evolutionary algorithm should offer better results than the previous naive one, so we explored its result by making some test runs using 32 individuals and 32 generations, double the number of generations we had used in our previous paper. How the fitness score evolves along all the generations is shown in Figure 1, which is an overview on how evolution proceeds. It shows many plateaus, first at score equal to 68 and then a big plateau for value = 62, to a point in which all the population has that score. However, exploration proceeds apace and eventually we obtain a ZAP score of 59 by the end of the evaluation budget.

From this initial exploration of values, we can conclude that even a small number of evaluations (only 1024) is able to obtain good results, and that the evolutionary algorithm is able to overcome, at least in some cases, plateaus with low fitness diversity across all the population. It is also evident that the evaluation budget is not enough and that more generations could be used to obtain better values of this score, down to 3 which seems to be the absolute minimum for the Juice Shop. This initial exploration took 6 hours in a Lenovo Carbon X5 laptop with a i7 CPU and Ubuntu 16.04, which also gave us an idea of the time we were going to need to devote to these experiments. The results of these will be shown next.

C. *Experimental results*

We performed several runs for population 16 and population 32, in every case with 32 generations. The main objective of these runs was not so much to measure the final result, since there are not so many evaluations, but to evaluate in which measure the evolutionary algorithm contributed to the improvement of the score of the generated configurations, as well as how many configurations, in the last generation, had the best score. We will examine individual results, shown in table IV. Experiments with population = 16 took around 6 hours in an Amazon EC2 instance, while experiments with population = 32 took twice as much; this is the main reason why no more results are shown. In practice, moving target defense would change configuration every few hours, which makes these results acceptable for its purpose, although it obviously would admit a certain degree of improvement.

The results in which population is only 16 evidence that what it essentially does is to generate different configurations with the best fitness found originally in the population: the final population is filled with mutated copies of a configuration, all of which have the same fitness. It happens to be 12 in these cases, which is a low ZAP score, but in the case an element with that score wouldn't have been in the initial population it would have been difficult to achieve that value with just a few evaluations (512, in this case). However, this result is acceptable, and shows that an evolutionary algorithm is able, at least, to generate a good amount of diverse configuration, even if at this population level it's not able to improve initial scores, just to weed out invalid configurations, or simply those with a low score.

The runs we were able to make with population = 32, and double the amount of evaluations we did before, 1024, do show a lot of improvement of initial configurations. In two cases, there's just one configuration with the same value, but most of them have a ZAP score below 62, which is a good value. In one case it was able to generate a quarter of the population with the same ZAP score, 53, but also a few more with values 54, 58 and 59, and all of them below 62, eliminating in any case all invalid configurations from the population. In all cases average is around 58, which is quite an improvement over initial averages, which are high mainly due to the presence of invalid configurations.

At any case, these results prove that improving the evolutionary algorithm makes our method able to extract many more valid configurations that can be used in the movable target defense method, and is able to do so in a reasonable amount of time; compared to our previous paper, the exploitation of values present in the initial population is better, and we are also able to improve initial values by some measure.

We are committed to open science and the reproducibility of results, which is why all the results of experiments, as well as their code and the scripts needed to generate this data can be found in https://github.com/JJ/2020-WCCI-variable-attack-surface, and can be reused with a free license.

IV. Conclusions and discussion

In this paper our main objective was to try and improve the evolutionary algorithm used for hardening and obtaining multiple configurations that can be used in a MTD policy. We tried first to use different options of the vulnerability scanning tool, finding them too costly to use, but also worse from the point of view of giving diverse scores to the configurations so that the evolutionary algorithm will work on them.

We then focused on working on a very limited evaluation budget, and used a 512 and 1024 evaluation evolutionary algorithm to try and generate a good set of configurations. In general, there are many "good" values that can be generated randomly for nginx configuration; however, to generate a diverse set of them with a low vulnerability score is more complicated. In this case, an evolutionary algorithm succeeded in finding that set, with 32 individuals being actually the minimum configuration that should be used in case we want to obtain configurations with a low average ZAP score; the few experiments we managed to do achieved a consistent score of around 58.

However, while these experiments are promising, and in fact deliver what we were looking for, diverse configurations in a

TABLE IV
EXPERIMENT RESULTS FOR EVERY RUN MADE FOR POPULATION 16 AND 32. "COPIES" INDICATES THE FRACTION OF THE POPULATION WHOSE VALUE IS THE SAME AS THE BEST INDIVIDUAL.

Population	Final.Best	Initial.Best	Copies	Initial.Avg	Final.Avg
16	12	12	1.00000	12.00000	448.5625
16	12	12	1.00000	12.00000	198.8125
16	12	12	1.00000	12.00000	320.9375
16	12	12	1.00000	12.00000	321.6875
32	3	3	0.03125	58.75000	391.3125
32	53	53	0.25000	58.59375	501.5625
32	51	53	0.03125	58.37500	473.8750

reasonable amount of time, they did reveal the need for faster evaluation, or simply another way of computing fitness.

This is why, in the immediate future, now that the evolutionary algorithm is working correctly, we will focus in trying to obtain the ZAP score faster. This is one of the main drawbacks of the algorithm right now, and although part of it is inherent, we could try to achieve faster evaluation by using a different web for generating the configuration and for deploying the configuration. Using the actual web for evolving configurations can be incredibly time-consuming; the use of surrogates would speed up evolution, either by creating surrogates of the web itself or by trying to make parts of the evaluation via surrogates found by using machine learning; this has been done already with CVSS scores [34] so it should be possible in principle. Simply working with implementation details might allow not only making things faster, but also doing parallel evaluation of several configurations at the same time, which right now is impossible due mainly to the fact that we're fixing the ports that are used for the websites we are evaluating.

But, more interestingly, we will try and expand the range of configurations we are using by going beyond initial requirements (used in STIG); this will expand the fitness landscape, so we might have to find a way of speeding up the evolutionary algorithm.

Finally, the evolutionary algorithm itself can be improved, by testing different types of selection procedures, and tuning its greediness. This is something that can be done immediately, and will be one of our next steps.

V. ACKNOWLEDGEMENTS

This paper has been supported in part by projects DeepBio (TIN2017-85727-C4-2-P).

REFERENCES

[1] NITRD, "NITRD CSIA IWG Cybersecurity Game-Change Research and Development Recommendations," https://bit.ly/2peOnfd, NITRD, May 2009.

[2] "National cyber leap year summit 2009 co-chairs report, networking and information technology research and development," September 2009.

[3] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving target defense: creating asymmetric uncertainty for cyber threats*. NY: Springer Science & Business Media, 2011, vol. 54.

[4] G.-l. Cai, B.-s. Wang, W. Hu, and T.-z. Wang, "Moving target defense: state of the art and characteristics," *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 11, pp. 1122–1153, Nov 2016. [Online]. Available: https://doi.org/10.1631/FITEE.1601321

[5] B. C. Ward, S. R. Gomez, R. Skowyra, D. Bigelow, J. Martin, J. Landry, and H. Okhravi, "Survey of cyber moving targets second edition," MIT Lincoln Laboratory Lexington United States, Tech. Rep., 2018.

[6] C. Lei, H.-Q. Zhang, J.-L. Tan, Y.-C. Zhang, and X.-H. Liu, "Moving target defense techniques: A survey," *Security and Communication Networks*, vol. 2018, p. 26 pp, 2018.

[7] A. Makanju, A. N. Zincir-Heywood, and S. Kiyomoto, "On evolutionary computation for moving target defense in software defined networks," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 287–288. [Online]. Available: http://doi.acm.org/10.1145/3067695.3075604

[8] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. ACM, 2019, pp. 469–484, event-place: Providence, RI, USA. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304037

[9] D. J. John, R. W. Smith, W. H. Turkett, D. A. Caas, and E. W. Fulp, "Evolutionary based moving target cyber defense," in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Comp '14. New York, NY, USA: ACM, 2014, pp. 1261–1268, event-place: Vancouver, BC, Canada. [Online]. Available: http://doi.acm.org/10.1145/2598394.2605437

[10] E. S. Collado, P. A. Castillo, and J. J. Merelo Guervós, "Using evolutionary algorithms for server hardening via the moving target defense technique," in *Applications of Evolutionary Computation - 23rd European Conference, EvoApplications 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15-17, 2020, Proceedings*, ser. Lecture Notes in Computer Science, P. A. Castillo, J. L. J. Laredo, and F. F. de Vega, Eds., vol. 12104. Cham: Springer, 2020, pp. 670–685. [Online]. Available: https://doi.org/10.1007/978-3-030-43722-0_43

[11] P. K. Manadhata and J. M. Wing, "A formal model for a systems attack surface," in *Moving Target Defense*. Springer, 2011, pp. 1–28.

[12] M. Piskozub, R. Spolaor, M. Conti, and I. Martinovic, "On the resilience of network-based moving target defense techniques against host profiling attacks," in *Proceedings of the 6th ACM Workshop on Moving Target Defense*, ser. MTD19. New York, NY, USA: Association for Computing Machinery, 2019, p. 112. [Online]. Available: https://doi.org/10.1145/3338468.3356825

[13] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *IEEE Security and Privacy*, vol. 12, no. 2, pp. 28–35, 2014, cited By 25. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84899582256

[14] J.-H. Cho, D. P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T. J. Moore, D. S. Kim, H. Lim, and F. F. Nelson, "Toward proactive, adaptive defense: A survey on moving target defense," 2019.

[15] J. Tian, R. Tan, X. Guan, Z. Xu, and T. Liu, "Moving target defense approach to detecting Stuxnet-like attacks," *IEEE Transactions on Smart Grid*, vol. 11, no. 1, pp. 291–300, 2019.

[16] B. Potteiger, Z. Zhang, and X. Koutsoukos, "Integrated moving target defense and control reconfiguration for securing cyber-physical systems," *Microprocessors and Microsystems*, vol. 73, p. 102954, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933119302364

[17] D. J. Prathyusha, S. Naseera, D. Anusha, and K. Alisha, "A review of biologically inspired algorithms in a cloud environment to combat DDoS attacks," in *Smart Intelligent Computing and Applications*. Springer, 2020, pp. 59–68.

[18] T. Shade, A. Rogers, K. Ferguson-Walter, S. B. Elsen, D. Fayette, and K. Heckman, "The Moonraker study: An experimental evaluation of host-based deception," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, 2020.

[19] E. Al-Shaer, "Toward network configuration randomization for moving target defense," in *Moving Target Defense*. Springer, 2011, pp. 153–159.

[20] V. Kansal and M. Dave, "Improving the effectiveness of moving target defenses by amplifying randomization," in *International Conference on Intelligent Computing and Smart Communication 2019*. Springer, 2020, pp. 27–36.

[21] T. Eghtesad, Y. Vorobeychik, and A. Laszka, "Deep reinforcement learning based adaptive moving target defense," 2019.

[22] J. Zheng and A. Namin, "A survey on the moving target defense strategies: An architectural perspective," *Journal of Computer Science and Technology*, vol. 34, no. 1, pp. 207–233, 2019, cited By 3. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85060727166

[23] S. X. Wu and W. Banzhaf, "The use of computational intelligence in intrusion detection systems: A review," *Applied Soft Computing*, vol. 10, no. 1, pp. 1 – 35, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494609000908

[24] A. Cui and S. J. Stolfo, "Symbiotes and defensive mutualism: Moving target defense," in *Moving target defense*. NY: Springer, 2011, pp. 99–108.

[25] Y. Huang and A. K. Ghosh, "Introducing diversity and uncertainty to create moving attack surfaces for web services," in *Moving target defense*. Springer, 2011, pp. 131–151.

[26] W. Peng, F. Li, C.-T. Huang, and X. Zou, "A moving-target defense strategy for cloud-based services with heterogeneous and dynamic attack surfaces," in *2014 IEEE International Conference on Communications (ICC)*. IEEE, 2014, pp. 804–809.

[27] M. Crouse and E. W. Fulp, "A moving target environment for computer configurations using genetic algorithms," in *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*. Piscataway, NY: IEEE, Oct 2011, pp. 1–7.

[28] B. Lucas, E. W. Fulp, D. J. John, and D. Cañas, "An initial framework for evolving computer configurations as a moving target defense," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*. NY, US: ACM, 2014, pp. 69–72.

[29] C. S. I. Group, "Common vulnerability scoring system version 3.1: Specification document," June 2019. [Online]. Available: https://www.first.org/cvss/specification-document

[30] X. Chi, J. Yao, and H. Yu, "A hybrid load balance method using evolutionary computing," in *Proceedings of the Australasian Joint Conference on Artificial Intelligence-Workshops*. NY, USA: ACM, 2018, pp. 15–19.

[31] DISA, "Apache server 2.4 UNIX security technical implementation guide (STIG)," DISA, Tech. Rep., January 2020. [Online]. Available: https://dl.dod.cyber.mil/wp-content/uploads/stigs/zip/U_Apache_Server_2-4_UNIX_STIG.zip

[32] B. Kimminich, "OWASP juice shop project," OWASP, Tech. Rep., 2020. [Online]. Available: https://www2.owasp.org/www-project-juice-shop/

[33] J. J. Merelo-Guervós, I. Blancas-Alvarez, P. A. Castillo, G. Romero, V. M. Rivas, M. G. Valdez, A. Hernandez-Aguila, and M. Román, "A comparison of implementations of basic evolutionary algorithm operations in different languages," in *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016*. IEEE, 2016, pp. 1602–1609. [Online]. Available: http://dx.doi.org/10.1109/CEC.2016.7743980

[34] M. Edkrantz and A. Said, "Predicting cyber vulnerability exploits with machine learning." in *SCAI*, 2015, pp. 48–57.