International Journal of Digital Crime and Forensics
Volume 14 • Issue 2

Security Enhancement Through Compiler-Assisted Software Diversity With Deep Reinforcement Learning

Junchao Wang, National Digital Switching System Engineering Technological R&D Center, Zhengzhou, China*

Jin Wei, School of Computer Science, Fudan University, Shanghai, China, Fudan University Data Arena Institute, Shanghai, China

Jianmin Pang, National Digital Switching System Engineering & Technological R&D Center, Zhengzhou, China

Fan Zhang, National Digital Switching System Engineering & Technological R&D Center, Zhengzhou, China

Shunbin Li, Zhejiang Lab, Hangzhou, China

ABSTRACT

Traditional software defenses take corresponding actions after the attacks are discovered. The defenders in this situation are comparatively passive because the attackers may try many different ways to find vulnerability and bugs, but the software remains static. This leads to the imbalance between offense and defense. Software diversity alleviates the current threats by implementing a heterogeneous software system. The N-Variant eXecution (NVX) systems, effective and applicable runtime diversifying methods, apply multiple variants to imporove software security. Higher diversity can lead to less vulnerabilities that attacks can exploit. However, runtime diversifying methods such as address randomization and reverse stack can only provide limited diversity to the system. Thus, the authors enhance the diversity of variants with a compiler-assisted approach. They use a deep reinforcement learning-based algorithm to generate variants, ensuring the high diversity of the system. For different numbers of variants, they show the results of the Deep Q Network algorithm under different parameter settings.

KEYWORDS

Deep Q Network, Multi-Compiling, N-Variant Execution, Software Diversity, Software Security, Variant Generation

I. INTRODUCTION

Software monoculture means that software systems adopt a relatively static fixed architecture and roughly similar operating mechanisms, which allows attacks that work on one system to be easily applied to all similarly configured systems. Most monoculture computer systems are subject to catastrophic failure in the event of a successful attack (Goth, 2003).

However, software diversity, a technical means that can effectively improve software security, has been a concern in recent years. Software systems with high diversity tend to resist more complex attacks, just as diverse biological populations form a safer and more stable ecosystem. This technology increases the cost and difficulty of attackers. With the continuous improvement of attack methods, the increasing variety of attack methods has led to an imbalance between offense and defense. A

DOI: 10.4018/IJDCF.302878 *Corresponding Author

This article published as an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/) which permits unrestricted use, distribution, and production in any medium, provided the author of the original work and original publication source are properly credited.

single software diversifying method often fails to achieve the expected security effect (Bansal, 2006; Massalin, 1987; Szekeres, 2013; Nergal, 2001; Bittau, 2014; Gras, 2017; Hund, 2013; Snow, 2013). Comparatively, N-variant eXecution (NVX) systems (Cox, 2006; Berger, 2006; Novark, 2010; Bruschi, 2007; Hosek, 2013, 2015; Kim, 2015; Kwon, 2016; Koning, 2016; Lu, 2018; Maurer, 2012; Salamat, 2009; Volckaert, 2015; Volckaert, 2016; Volckaert, 2012; Xu, 2017; Osterlund, 2019; Voulimeneas, 2020; Wu, 2020) are more effective and adaptable. The work of Franz (2018) mentioned that NVX technology tries to make a system achieve probabilistic security by introducing redundant variants. The more considerable diversity between variants will make the system safer. The implementation principle of this technology is to run different variants of the same program simultaneously and watch the behaviors of these variants. The attackers need to simultaneously destroy multiple program variants without causing system errors. In recent research, NVX systems have been widely used in systems with high security requirements, but there are still some problems in existing NVX systems. Most NVX systems only use runtime diversifying approaches, such as the disjoint code layout (DCL) (Volckaert et al., 2015), reverse stack (Salamat, 2009), and address randomization (Berger, 2006; Lu, 2018), which only provide limited diversity. Thus, they are often incapable of attacks using program structures or data flows, such as position-independent return-oriented programming (PIROP) attacks (Göktas et al., 2018) and certain data-oriented programming (DOP) attacks (Hu et al., 2016).

Our approach proposed in this paper solves the above problem. We increase the diversity of the NVX systems by using a multicompiling method and propose a compilation tree model to evaluate the variants' diversity. In addition, we present a variant generation algorithm based on deep reinforcement learning, which is verified in the experimental part.

Applying multicompiling methods can increase the variants' diversity and maintain their functional equivalence. The traditional obfuscation methods used by attackers to protect them from anti-virus software can also be applied for software protection. Thus, we can achieve multicompiling on the source code of software by using existing obfuscation tools. The current obfuscation tools provide many obfuscation approaches, so we can establish NVX systems with great diversity with these existing obfuscation options. To quantitatively measure the diversity of variants, we propose a compilation tree method. In our model, the tree's nodes represent the variants with different compiling methods, and the distances between the nodes represent the difference between the variants. In this way, the problem of diversity measurement is transformed into calculating the distance between tree nodes. The optimization goal of our algorithm is to maximize the variants' diversity. We apply a brute force algorithm and deep Q-learning network (DQN) (Mnih et al., 2013) algorithm to generate variants. We also verify in the experiments that the DQN method is more practical than the brute force method when the number of variants is large.

The contributions of this paper are as follows:

- We increase the diversity of variants by using multicompiling methods to enhance the security of the NVX systems.
- We propose a compilation tree model to represent the differences between the compiled variants. The advantage of this method is that the differences between variants can be calculated through the tree structure, which provides a basis for the variant generation algorithm.
- We propose a variant generation algorithm that can be applied in NVX systems. We use a brute force algorithm and DQN algorithm to generate the combination of the most secure variant.
- We verify the applicability of our algorithm in the experiment. For different numbers of multicompiling methods and required variants, we compare the diversity of variant sets and analyze the DQN algorithm under different settings.

II. RELATED WORKS

We list the works related to our approach in this section. We find that most of the NVX systems use runtime diversifying approaches, and we summarize the multicompiling methods that can be used to enhance the diversity in the NVX systems. Moreover, compared with the existing variants' diversity measuring methods, we quantitatively analyze the diversity that our method brings to the system from the compilation perspective.

A. NVX SYSTEMS

Since 2006, many NVX systems have been used in security scenarios, such as memory security and kernel security. The variants of NVX run on the same (Cox, 2006; Berger, 2006; Novark, 2010; Bruschi, 2007; Hosek, 2013, 2015; Kim, 2015; Kwon, 2016; Koning, 2016; Lu, 2018; Maurer, 2012; Salamat, 2009; Volckaert, 2015; Volckaert, 2016; Volckaert, 2012; Xu, 2017; Osterlund, 2019) or different (Voulimeneas, 2020; Wu, 2020) physical machines simultaneously. They provide the same incentives to the variants and compare the variants' output behavior through a monitor. The idea of NVX systems has progressed rapidly in recent years. The authors of Berger and Zorn (2006), Novark and Berger (2010) built Diehard and DieHarder architecture and probabilistic analysis to protect memory security. The work of Kim et al. (2015) and Kwon et al. (2016) reduced the uncertainty of offline comparison by comparing two variants in real time. The work of Osterlund et al. (2019) protected kernel security through NVX systems. Moreover, Cox et al. (2006) first proposed the N-variant system in 2006 and compared several variant generation strategies, such as disjoint layout mapping and instruction set randomization. Salamat et al. (2009) proposed Orchestra, using reverse stack growth to generate stacks in opposite directions. GHUMVEE (Volckaert, 2015; Volckaert, 2016; Volckaert, 2012) applied disjoint code layouts (DCLs) to introduce software diversity. This method ensured no segment overlap in the address space of code variants so that there were no coexisting gadgets during return-oriented programming (ROP) attacks. In 2015, Hosek and Cadar (2015) proposed Varan, which relied on static binary tools to significantly improve the NVX systems' performance, focusing more on software reliability than security. Similarly, Mx (Hosek & Cadarl, 2013) and Taychon (Maurer & Brumley, 2012) also focused more on the system's performance. As a new NVX system, MvArmor (Koning et al., 2016) realized a high-performance NVX system through hardware assistance and virtualized processes, and they put the software in user-level privilege instead of kernel-level privilege to protect the security of the monitor. In summary, existing NVX systems use runtime diversifying approaches to introduce limited diversity.

B. VARIANTS' DIVERSITY MEASURING METHODS

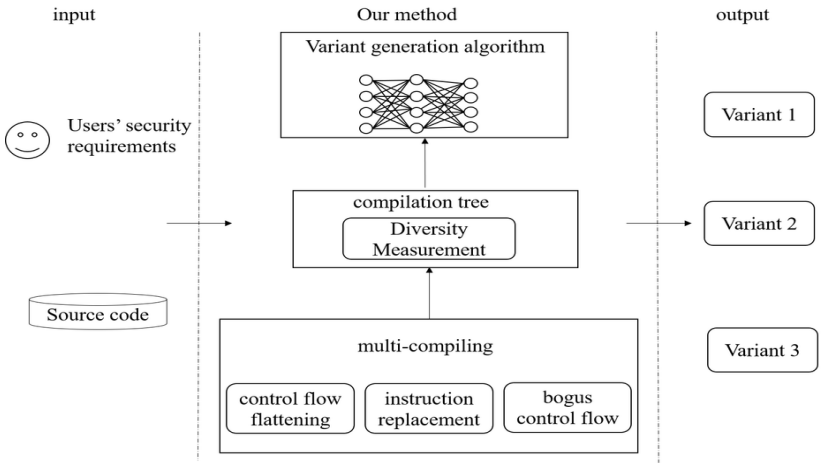
Homescu et al. (2017) mentioned that entropy and granularity are two critical indicators for variant systems. The two indicators are large enough so that the attacker must spend many resources to brute the force of the crack. Measuring the variants' diversity. The authors of Liu et al. (2018), Gu et al. (2017), and Tong et al (2019) divided variants into components such as processor, operating system, application software, protocol stack, etc., and measured the differences of components separately. From a security perspective, the works of Zhang et al. (2020), and Li et al. (2018) used symbiosis vulnerabilities to compare the similarity between the two variants. The more common vulnerabilities contained in the two variants, the smaller the diversity between the two variants. The authors of Liu et al. (2018), and Gu et al. (2017) evaluated variant diversity by measuring the heterogeneity or safety of the variant system. The work of Zhang et al. (2019) also considered the heterogeneity and service quality of the generated variants. Software diversity can be easily introduced from the compiling process, but we do not find any existing work evaluating the variants' diversity from the perspective of compilation.

C. MULTICOMPILING METHODS

Multicompiling can be regarded as a program compiling algorithm that takes a particular program as input, and the output is a functionally equivalent program but more challenging to understand. The work of Banescu and Pretschner (2018) lists many compiling techniques currently in use, including opaque predicate, variable division/merging, control flow flattening, instruction replacement, garbage code injection, virtualization obfuscation, etc. The current multicompiling methods have been integrated into automation tools. For C language, for example, obfuscator-llvm (OLLVM)[1], Hikari[2], Armariris[3], Tigresss[4], etc., they mainly support control flow flattening, instruction replacement, bogus control flow, and other means. These tools are easy to transplant to other tools. Multicompiler[5] supports more obfuscation methods than other tools, including code randomization, such as function sorting, CPU register variable allocation, instruction scheduling, insertion of NOP instructions and instruction replacement; stack layout randomization, such as reordering stack elements, filling stack elements and frames; global variables randomizing, such as rearranging the order of global variables, randomly adding padding values to destroy global variables attack. Additionally, ProGuard[6] can obfuscate the files in the format of jars, aars, wars, ears, zips, apks, or directories; PyArmor[7] uses the encrypted command to achieve the purpose of protection; Obfuscator.io[8] supports obfuscation of JavaScript; Yakpro PO[9] supports obfuscation of PHP language; SharpLoader[10] supports obfuscation of C# language.

Multicompiling methods always have many applications in computer security. Jacob et al. (2008) proposed the idea of a "superdiversifier", a super performance compiler aimed at improving computer security. Giuffrida et al. (2012) used multicompiling methods to protect kernel-level system security by transforming the layout of code and data. Their code conversion mainly included function shuffling and reordering of basic blocks in functions. Homescu et al. (2017) proposed a compiler-based software automatic diversification technology by considering the two indicators of entropy and granularity. Two obfuscation methods were discussed: insertion of NOP instructions and instruction scheduling. The vast number of multicompiling options can provide us with enough diversifying options to build a compilation tree.

Figure 1. Framework of our method



III. COMPILATION TREE

A. Compilation Tree Framework

As shown in Figure 1, our method's input is the source code of a software and users' security requirements, and the output is a set of compiled binaries named variants. Based on the heterogeneous variants generated by multicompiling methods, we can build a robust software diversity system. To compile the source code of the software, we can use the different compiling options provided by existing compiling tools, such as control flow flattening, variable division, instruction replacement, and bogus control flow. We can also use different parameters in the same method or a combination of different compilation methods. Thus, we propose a compilation tree model to represent the differences between different variants. We also apply a DQN algorithm to generate variants; this part is introduced in detail in section IV.

The notation in our model and their description are listed in Table 1.

Table 1. Notation and their descriptions in our algorithm

Algorithm1: Variants Generation method based on brute force algorithms

Input: Number of nodes m, Number of needed variants k
step1: Calculate the distance between every two variants
for n_i, n_j do
  calculate $d(n_i, n_j)$
  end for
step2: output the maximum diversity and corresponding variant combination
Initialize: $d_v = 0$
Initialize set V with k variants, $V = \{n_1, n_2, ..., n_k\}$
while (true) do
  if All k variants combinations are calculated
    return d_v, V
  else
    calculate the diversity of V^{new} according to Eq.4
    if $dv^{new} > dv$
      $dv \leftarrow dv^{new}$
      $V \leftarrow V^{new}$
  return dv, V
Output:dv, V

We define the following terms and their meanings in our proposed compilation tree model.

Definition 1 node v_i: The *i*-th node in the tree, representing a variant generated by using a specific compiling method or several compiling methods.

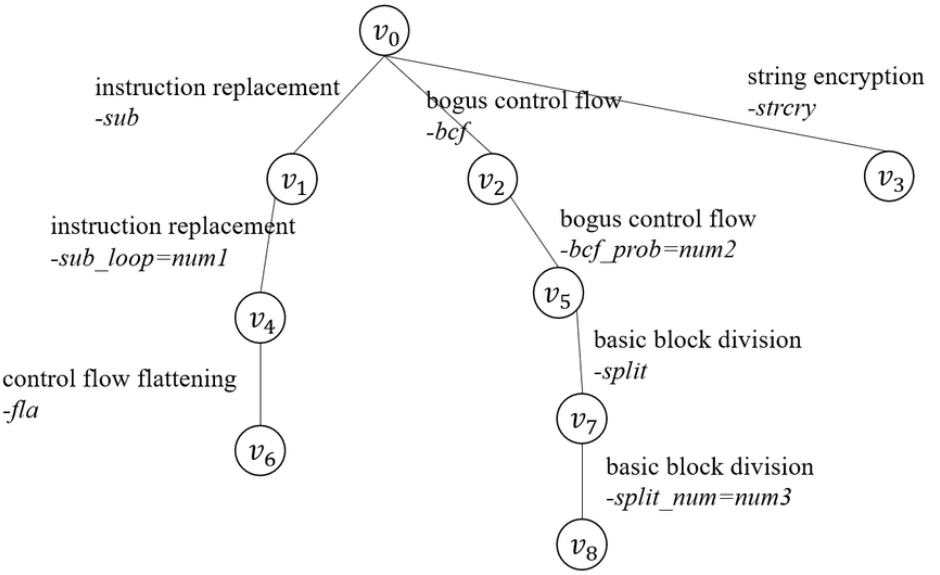
Definition 2 root node v_0: A node without a predecessor node in a nonempty tree, representing a variant that does not use the multicompiling method to compile.

Definition 3 edge: the edges connect two adjacent nodes.

Definition 4 edge weight d (v_i, v_j): the weight of edges, used to represent the distance between the two variants connected by the edge. For adjacent nodes v_1 and v_2, the distance between them is d (v_1, v_2). It should be noted that in the compilation tree, any two nodes cannot be generated with

the same multicompiling methods. This is mainly because the same multicompiling methods cannot introduce diversity.

Figure 2. Compilation tree realized by compiling methods



Our compilation tree model is shown in Fig. 2. v_0 is the root node, meaning the source code of software; $v_1 \sim v_8$ denote the variants compiled by the single or combined compiling methods. The edges of the tree model represent several compiling methods, including instruction replacement, bogus control flow, and string encryption. In this tree structure, the edge weights represent the distance between the two connected nodes.

When using multicompiling options to achieve different code obfuscation levels, we need to define the parameters of multicompiling options. For example, in OLLVM, the instruction replacement option can be expressed as "*-sub*" or "*-sub_loop=num1*". "*-sub*" means to perform instruction replacement with the default value. "*-sub_loop=num1*" means that the number of replacement loops is *num1*. *num1* can be any integer, and the default value of *num1* is 1. The bogus control flow option is to add a basic block before the current basic block to modify the function call graph or fill the original basic block with randomly selected garbage commands. This option can be expressed as "*-bcf*" or "*-bcf_prob=num2*". "*-bcf*" means to perform bogus control flow with the default value. "*-bcf_prob=num2*" means this method covers *num2* percent blocks. Therefore, each method's parameters can also introduce diversity to the variants. As an example, nodes v_1, v_2, v_3, v_4 and v_5 represent the variants realized by the single compiling methods. v_1 and v_4 indicate the variants generated using the instruction replacement method. Since *num1* can be any number except the default value, v_1 and v_4 are also different. Similarly, the number of blocks applied by the bogus control flow command is different, so that v_2 and v_5 are different. The distance between v_4 and v_5 can be expressed in Eq.

$$d\left(v_4, v_1\right) + d\left(v_1, v_0\right) + d\left(v_0, v_2\right) + d\left(v_2, v_5\right) \tag{1}$$

When using two or more methods to compile the variants, we find that it can produce a strong effect, and there are many combined compilation technologies in many practice to improve the security of the variants' environment (Naumovich et al., 2006; Forte et al., 2017) Nodes v_6, v_7 and v_8 represent the variants compiled by the combined compiling methods. Starting from the root node v_0, reaching the v_6 node needs to pass through v_1 and v_4, which indicates that v_6 is obtained after v_4 implementing control flow flattening (*-fla*). In the same way, to reach node v_7, we first need to use bogus control flow to compile the variants and then apply basic block division (*-split)* to the variant. The difference between v_7 and v_8 is the difference in the divided instruction blocks. The distance between v_6 and v_8 is presented in Eq. 2, where $d\left(v_4, v_5\right)$ can be calculated by Eq.

$$d\left(v_6, v_8\right) = d\left(v_6, v_4\right) + d\left(v_4, v_5\right) + d\left(v_5, v_8\right) \tag{2}$$

B. Problem Formulation

Our purpose is to solve the problem of variant generation to achieve the maximum diversity. Here are a few terms that we will use in our optimization problem.

Definition 5 set *V*: The variant set is generated by our method, which is also the output variant combination in Figure 1.

Definition 6 variants' diversity *dv*: the diversity of variants set.

Definition 7 selecting factor x_i : x_i indicates whether variant v_i is included in variant set V generated by our algorithm.

$$x_i = \begin{Bmatrix} 1, & x_i \in V \\ 0, & x_i \notin V \end{Bmatrix} \tag{3}$$

It is worth noting that the distance between two nodes v_i and v_j is the sum of weights of edges along the shortest path between two nodes. In our model, there is no variant of the repeated compiling method in the generated variant set *V*. We can define our optimal function as:

$$\max dv = \sum_{i=1}^{m}\sum_{i=1}^{m} x_i x_j d\left(v_i v_j\right), i \neq j \tag{4}$$

m is the number of nodes in the compilation tree. This formula means that the diversity of set *V* is the sum of distances between nodes in *V*.

The simplest way to solve this optimization problem is to traverse each combination of variants to find the node set with the largest d_k. The time complexity of this brute forcing method is

$O(C_m^k) = O(\frac{m!}{k!(m-k)!})$. Therefore, the size of the search space is related to the compilation trees' scale and the number of required variants.

Algorithm 1

notation	description
v_i	i^{th} variant in the compilation tree
$d(v_i, v_j)$	the distance between v_i and v_j
m	number of nodes in the compilation tree
k	number of variants required by the user
V	the set of generated variants
dv	the diversity of variants set V
state	each state corresponds to a variants selection method
d_{state}	diversity of variants in a state
action	the action taken for the current state
reward	the benefit after taking an action
γ	discount factor
lr	learning rate

Algorithm 2

Algorithm2: Variants Generation Method based on DQN

Input: Number of nodes m, Number of needed variants k, discount factor γ, episode
step1: Training the DQN with weight θ
Initialize: the weights of edges in compilation tree
Initialize: action-value function Q with 0
Initialize: replay memory D
for i = 1, episode **do**
  Initialise sequence with k variats, $s = \{n_1, n_2, ..., n_k\}$
  for t = 1, T **do**
    select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t and get s_{t+1}
    observe $reward = d(s_{t+1}) - d(s_t)$
    store transition$(s_t, a_t, s_{t+1}, reward)$ in D
    Sample random minibatch of transitions $(s_t, a_t, s_{t+1}, reward)$ from D
    set $y = reward + \gamma \max_{a'} Q(s^{'}, a^{'}; \theta_{i-1})$
    perform a gradient descent step on $(y - Q(s, a, \theta))^2$
    end for
  end for
step2: predict set V
Initialize set V with k variats, $V = s_0^{'} = \{n_1, n_2, ..., n_k\}$
Initialize $t = 0$
while the stopping criterion is not satisfied **do**
  select $a_t = \max_a Q^*(s_t^{'}, a; \theta)$
  Execute action a_t and get $s_{t+1}^{'}$
  $V \leftarrow s_{t+1}^{'}$
  calculate dv for set V
  end while
Output: V, dv

IV. OUR ALGORITHM

For a given number of variants *k*, our algorithm's goal is to maximize the diversity of the system. The security of the system is directly proportional to the variants' diversity. To achieve this goal, we need to select the most diverse combination of variants. Therefore, there are two difficulties that need to be solved:

1) How to quantify the variants' diversity?
2) How can the most diverse combination of variants be generated?

For the first problem, we use the distance of the variant's combination to define the variants' diversity. For the second problem, we introduce two variant generation algorithms: the brute force method and the DQN method.

A. Variant Generation Method Based on Brute Force Algorithms

For a compilation tree, all nodes can be regarded as a method of variant generation. Therefore, in the brute force algorithm, we permutate all possible combinations and find the most diverse combination of variants. By using Eq. 4 to calculate the variants' diversity, this process is presented in Algorithm 1.

B. Variants Generation Method based on DQN Algorithm

Q-learning is a typical algorithm for value-based solutions in reinforcement learning (Sutton & Barto, 2018). We can use the Q-table or Q-function to store the action-value function (Q value) corresponding to each state action to make the next action selection. $Q*(s,a)$ can be defined as the optimal action-value function, which means the maximum expected return achievable by any following strategy, in state *s* for action a. The optimal action-value function obeys the Bellman equation:

$$Q^*\left(s,a\right) = E\left[reward + {}^{3}\max_{a'} Q^*\left(s'a'\right)\right] \tag{5}$$

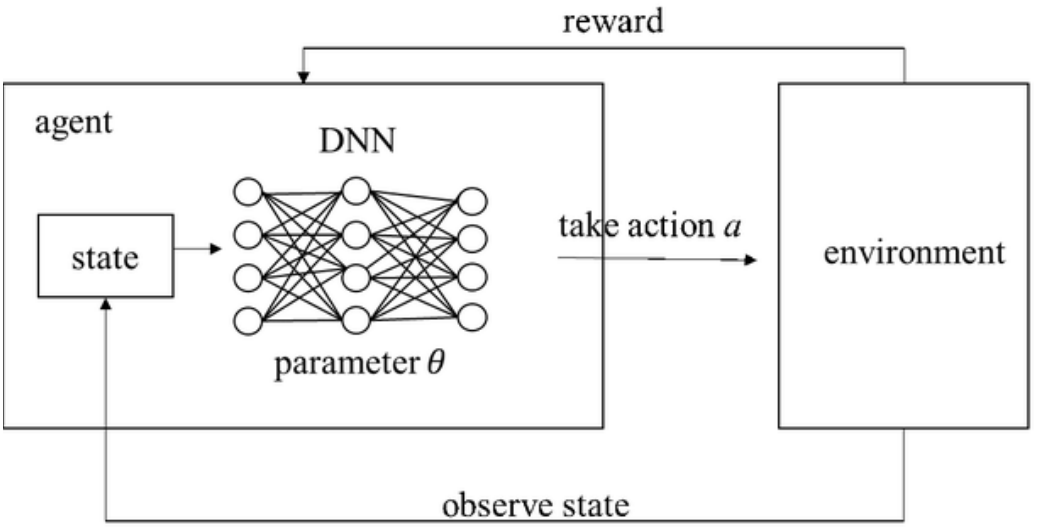
y is the discount factor, and $\max_{a'} Q_i(s',a')$ means the optimal value in the next state for all possible actions a'

We can use an iterative update for the Bellman equation:

$$Q_{i+1(s,a)} = E\left[reward + \gamma \max_{a'} Q^*\left(s'a'\right)\right] \tag{6}$$

It can be proved that $Q_i \to Q* as\ i \to \infty$ (Sutton & Barto, 1998).

Figure 3. Schematic diagram of DQN algorithm



The work of Mnih et al. (2013) proposed the deep Q network (DNN), which combines a deep neural network (DNN) (Miikkulainen et al., 2019) and Q-learning. As shown in Fig. 3, the input of the DNN with weight θ is the current state, and the output is an action that can be selected. The environment gives the corresponding reward value to guide the agent to complete a specific purpose or make a profit maximized action. The DQN algorithm also encourages many algorithms that combine DNN and Q-learning, such as the double deep Q network (DDQN) (Hasselt et al., 2016), Du Lin DQN (Wang et al., 2019), Noisy DQN (Fortunato et al., 2017), and Rainbow (Hessel et al., 2018).

Algorithm 1 is not applicable when the number of nodes is enormous. Thus, we apply the DQN to generate variants. In our algorithm, the state space is discrete. When training the neural network, to train more variant combinations, we map the entire state space into a one-dimensional array. The input to the neural network is a state, which is a combination of variants of the required number. Our network starts with two 100-way fully connected layers with *the ReLU* function and ends with a 3-way fully connected layer with a linear function. We define the symbols used in the DQN algorithm as follows:

Definition 8 node combination: a combination of *k* nodes, where *k* is the number of variants required by the user.

Definition 9 state *s:* each node combination corresponds to a state. s_i is the i-th state in state array.

Definition 10 action *a*: selecting different node combinations in the state space array, the corresponding action values are different. The new node combination is randomly selected, and each variant in the combination may be different from the previous state. Suppose the current state is $\mathrm{s_i}$, and the state after selecting the variant in the next step is $\mathrm{s_j}$. Then, we can define the value of action:

$$action = \begin{Bmatrix} 0, & j = i \\ 1, & j = i - i \neq 0 \\ 2, & j = i \end{Bmatrix} \tag{7}$$

Definition 11 reward: the reward score given by the environment when taking different actions. Assuming that the current state is s_i and the state obtained after taking action is s_j, the expression of reward is represented as follows:

$$reward = d\left(s_i\right) - d\left(s_j\right) \tag{8}$$

The loss function is defined as:

$$L\left(\theta\right) = \left(reward + \gamma \max_{a'} Q^*\left(s'a';\theta\right) - Q\left(s'a';\theta\right)\right) \tag{9}$$

y is the discount factor; the larger the value is, the more critical it is to historical experience. When y=0, only the current benefit (reward) is considered; is the optimal value that can be obtained in the next state.

We perform a gradient descent step on the equation to update the weight ¸ . Moreover, in Eq. 9, we choose off-policy temporal-difference (TD) learning (Sutton & Barto, 1998) to update the Q-value as follows:

$$Q\left(s'a';\theta\right) = \left(1 - lr\right)Q\left(s,a;\theta\right) + lr\left[reward + \gamma \max_{a'} Q^*\left(\left(s'a';\theta\right)\right)\right] \tag{10}$$

where $ir \in (0,1)$ in Eq.10 is the learning rate. The TD algorithm is the central and novel part of reinforcement learning, which can update the knowledge of the agent at every timestep. It contains two policies: on-policy TD control (Sarsa) and off-policy TD control (Q-learning).

Referring to work (Mnih et al., 2013), we introduce the variant generation method illustrated in Algorithm 2.

V. EXPERIMENTS

A. Experimental Setup and Dataset

The experiments are performed on a server with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz 16-core CPU and 250 GB RAM.

We analyze different factors that can affect variant generation. Thus, we benchmark the performance of our proposed algorithm under different settings. To evaluate the experimental data, we randomly generate a tree structure with different numbers of node and edge weights.

B. Experimental Results

1) different numbers of compiling options

To compare the impact of different compilation options on the experimental results, we generated compilation trees with 10, 50, 100, 500, and 1000 nodes. In our model, each node represents a compilation option.

The following experimental results are all obtained under the condition of the required number of variants *k=3*. The brute force and the DQN algorithms use the same tree model.

Table 2 shows the experimental results of the brute force algorithm.

Table 2. The results of the brute force algorithm

nodes number m	10	50	100	500	1000
diversity d_v $(k=3)$	62	100	106	174	194

Figure 4, Figure 5, and Figure 6 show the effect of the learning rate, discount factor, and episode on the DQN algorithm result. The x-axis is the number of nodes in the compiled tree. As the number of nodes increases, the diversity of the generated variants increases.

Figure 4. The results of different learning rate in the DQN algorithm

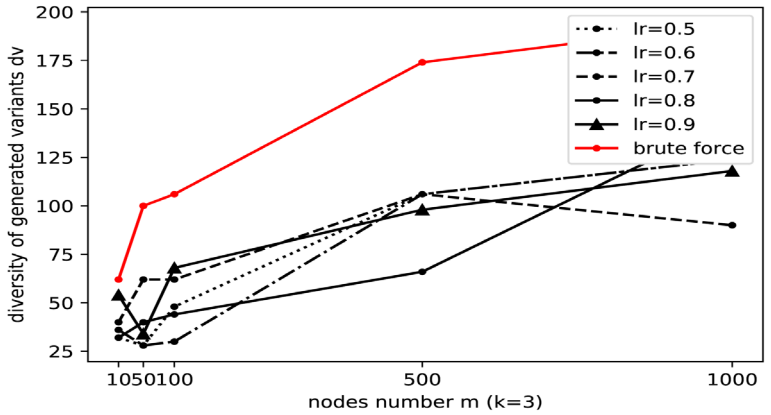
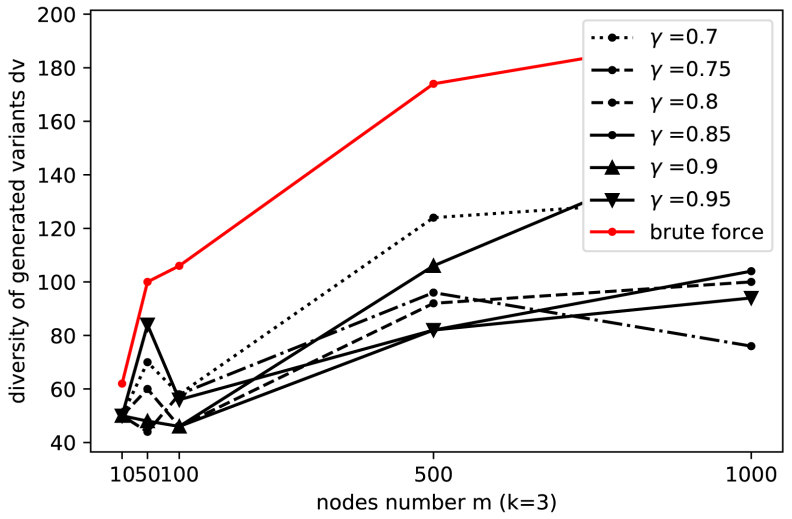


Figure 5. The results of different discount factor in the DQN algorithm



In Figure 4, the red line represents the brute force algorithm's experimental results, and the five black lines represent the diversity of generated variants when the learning = 0.5, 0.6, 0.7, 0.8, and 0.9. The default factor = 0.95; episodes = 100. The value of the learning rate affects the experimental results. When the learning rate is equal to 0.7, the generated variants' diversity is the largest.

The six black lines in Figure 5 represent the diversity of generated variants when the discount factor Y takes different values. The default parameters are lr = 0.8 and episodes = 100. When the number of nodes is small, the variants' diversity is largest when Y= 0.7, and when the number of variants is large, Y= 0.9 is better.

Figure 6. The results of different episode in the DQN algorithm

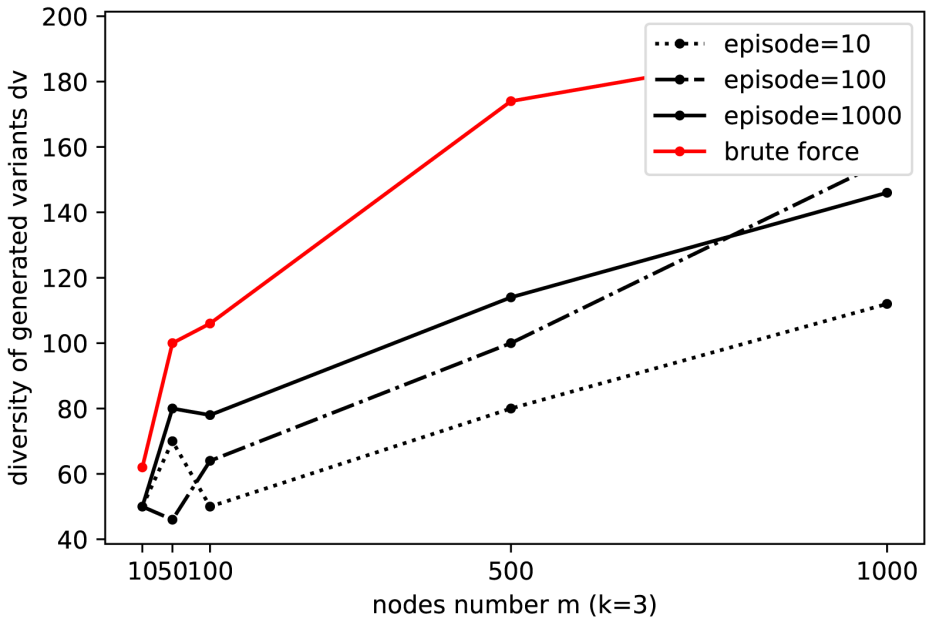
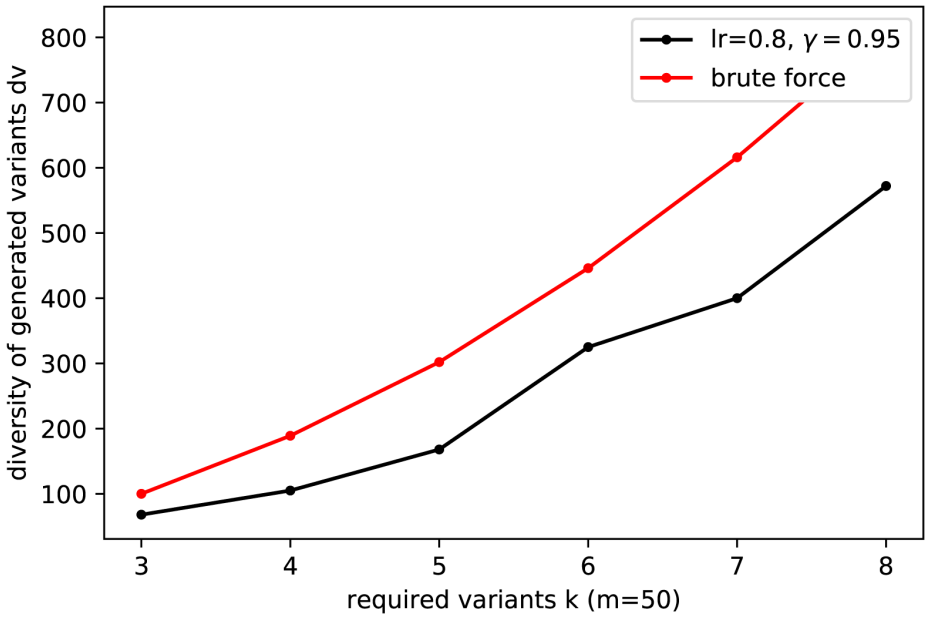


Figure 6 shows that when the number of episodes is set to 1000, the generated variant combination has greater diversity, and larger training times can obtain better experimental results.

2) different numbers of required variants

When the number of nodes is 50, we compare the impact of different required variant numbers *k* on the generated variant diversity. As shown in Figure 7, the parameters of the DQN algorithm are . When *k* increases, the gap of diversity between the DQN algorithm and the brute force decreases, and our algorithm has strong applicability. We can also conclude that as the number of required variants increases, the diversity of generated variants will also increase.

Figure 7. The results of different required variants numbers



It can be seen from the experimental results that if the cost permits, we can appropriately increase the number of compilation options and required variants to obtain high diversity. In addition, although the DQN algorithm cannot obtain the optimal solution every time, it is more efficient and can achieve better results when the number of variants is large.

VI. CONCLUSION

In this paper, we enhance the diversity of NVX systems by using the multicompiling method. We present a compilation tree model to measure the variants' diversity and apply the DQN algorithm to generate variants. Our variant generation algorithm ensures the high diversity of the NVX systems to improve the system's resistance to attacks. In the experiment, we compare the results of the DQN algorithm with different settings and analyze the impact of variant numbers and compiling option numbers on variant diversity.

In future work, we will generate the weights of the edges in the compilation tree based on techniques such as binary diffing analysis. We will also consider system performance and more factors when generating variants.

VII. ACKNOWLEDGMENTS

This research is supported by the National Key Research and Development Program of China (2017YFB0803202), Major Scientific Research Project of Zhejiang Lab (No. 2018FD0ZX01), National Core Electronic Devices, High-end Generic Chips and Basic Software Major Projects (2017ZX01030301), the National Natural Science Foundation of China (No. 61309020) and the National Natural Science Fund for Creative Research Groups Project (No. 61521003).

REFERENCES

Banescu, S., & Pretschner, A. (2018). A tutorial on software obfuscation. *Advances in Computers*, *108*, 283–353.

Bansal, S., & Aiken, A. (2006). Automatic generation of peephole superoptimizers. *ACM SIGARCH Computer Architecture News*, *34*(5), 394–403. doi:10.1145/1168919.1168906

Berger, E. D., & Zorn, B. G. (2006). DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, *41*(6), 158–168. doi:10.1145/1133255.1134000

Bruschi, D., Cavallaro, L., & Lanzi, A. (2007, April). Diversified process replicæ for defeating memory error exploits. In 2007 IEEE International Performance, Computing, and Communications Conference (pp. 434-441). IEEE.

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., & Hiser, J. (2006, August). N-Variant Systems: A Secretless Framework for Security through Diversity. In *USENIX Security Symposium* (pp. 105-120).

Forte, D., Bhunia, S., & Tehranipoor, M. M. (Eds.). (2017). *Hardware protection through obfuscation*. Springer International Publishing.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., & Legg, S. (2017). *Noisy networks for exploration*. arXiv preprint arXiv:1706.10295.

Franz, M. (2018). Making multivariant programming practical and inexpensive. *IEEE Security and Privacy*, *16*(3), 90–94.

Franz, M. (2018). Making multivariant programming practical and inexpensive. *IEEE Security and Privacy*, *16*(3), 90–94.

Giuffrida, C., Kuijsten, A., & Tanenbaum, A. S. (2012). Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium (USENIX Security 12)* (pp. 475-490). USENIX.

Göktas, E., Kollenda, B., Koppe, P., Bosman, E., Portokalidis, G., Holz, T., & Giuffrida, C. (2018, April). Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (pp. 227-242). IEEE.

Goth, G. (2003). Addressing the monoculture. *IEEE Security and Privacy*, *1*(6), 8–10. doi:10.1109/MSECP.2003.1253561

Gu, Z. Y., Zhang, X. M., & Lin, S. j. (2017). Research on load-aware dynamic scheduling mechanism based on security strategies. *Jisuanji Yingyong*, *37*(11), 3304–3310.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. *Thirty-second AAAI conference on artificial intelligence*.

Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., & Franz, M. (2015). Large-scale automated software diversity—Program evolution redux. *IEEE Transactions on Dependable and Secure Computing*, *14*(2), 158–171. doi:10.1109/TDSC.2015.2433252

Hosek, P., & Cadar, C. (2013, May). Safe software updates via multi-version execution. In 2013 35th International Conference on Software Engineering (ICSE) (pp. 612-621). IEEE. doi:10.1109/ICSE.2013.6606607

Hosek, P., & Cadar, C. (2015). Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Computer Architecture News*, *43*(1), 339–353. doi:10.1145/2786763.2694390

Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., & Liang, Z. (2016, May). Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)* (pp. 969-986). IEEE.

Jacob, M., Jakubowski, M. H., Naldurg, P., Saw, C. W. N., & Venkatesan, R. (2008, November). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security* (pp. 100-120). Springer.

Kim, D., Kwon, Y., Sumner, W. N., Zhang, X., & Xu, D. (2015). Dual execution for on the fly fine grained execution comparison. *ACM SIGARCH Computer Architecture News*, *43*(1), 325–338. doi:10.1145/2786763.2694394

Koning, K., Bos, H., & Giuffrida, C. (2016, June). Secure and efficient multi-variant execution using hardware-assisted process virtualization. In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 431-442). IEEE. doi:10.1109/DSN.2016.46

Kwon, Y., Kim, D., Sumner, W. N., Kim, K., Saltaformaggio, B., Zhang, X., & Xu, D. (2016, March). Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 503-515). doi:10.1145/2872362.2872395

Li, W., Zhang, Z., Wang, L., & Wu, J. (2018). The modeling and risk assessment on redundancy adjudication of mimic defense. *Journal of Cyber Security*, *3*(5), 64–74.

Lu, K., Xu, M., Song, C., Kim, T., & Lee, W. (2018). Stopping memory disclosures via diversification and replicated execution. *IEEE Transactions on Dependable and Secure Computing*.

Massalin, H. (1987). Superoptimizer: A look at the smallest program. *ACM SIGARCH Computer Architecture News*, *15*(5), 122–126. doi:10.1145/36177.36194

Maurer, M., & Brumley, D. (2012). TACHYON: Tandem execution for efficient live patch testing. In 21st {USENIX} Security Symposium ({USENIX} Security 12) (pp. 617-630).

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., & Hodjat, B. (2019). Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing* (pp. 293–312). Academic Press.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning*. arXiv preprint arXiv:1312.5602.

Naumovich, G., Yalcin, E., Memon, N. D., Yu, H. H., & Sosonkin, M. (2006). *U.S. Patent No. 7,150,003.* Washington, DC: U.S. Patent and Trademark Office.

Novark, G., & Berger, E. D. (2010, October). DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (pp. 573-584).

Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., & Giuffrida, C. (2019, April). kMVX: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 559-572). doi:10.1145/3297858.3304054

Qinrang, L. I. U., Senjie, L. I. N., & Zeyu, G. U. (2018). Heterogeneous redundancies scheduling algorithm for mimic security defense. *Journal of Communication*, *39*(7), 188.

Salamat, B., Jackson, T., Gal, A., & Franz, M. (2009, April). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems* (pp. 33-46). doi:10.1145/1519065.1519071

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., & Sadeghi, A. R. (2013, May). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy* (pp. 574-588). IEEE. doi:10.1109/SP.2013.45

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction.* MIT Press.

Szekeres, L., Payer, M., Wei, T., & Song, D. (2013, May). Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy* (pp. 48-62). IEEE. doi:10.1109/SP.2013.13

Thrun, S., & Littman, M. L. (2000). Reinforcement learning: An introduction. *AI Magazine*, *21*(1), 103–103.

Tong, Q., Guo, Y., Hu, H., Liu, W., Cheng, G., & Li, L. S. (2019). A Diversity Metric Based Study on the Correlation between Diversity and Security. *IEICE Transactions on Information and Systems*, *102*(10), 1993–2003.

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*(1).

Volckaert, S., Coppens, B., & De Sutter, B. (2015). Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, *13*(4), 437–450. doi:10.1109/TDSC.2015.2411254

Volckaert, S., Coppens, B., Voulimeneas, A., Homescu, A., Larsen, P., De Sutter, B., & Franz, M. (2016). Secure and efficient application monitoring and replication. In 2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16) (pp. 167-179).

Volckaert, S., De Sutter, B., De Baets, T., & De Bosschere, K. (2012, October). GHUMVEE: efficient, effective, and flexible replication. In *International Symposium on Foundations and Practice of Security* (pp. 261-277). Springer, Berlin, Heidelberg.

Voulimeneas, A., Song, D., Parzefall, F., Na, Y., Larsen, P., Franz, M., & Volckaert, S. (2020, June). Distributed heterogeneous n-variant execution. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 217-237). Springer, Cham. doi:10.1007/978-3-030-52683-2_11

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995-2003). PMLR.

Wojtczuk, R. (2001). The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e, 70. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., & Boneh, D. (2014, May). Hacking blind. In 2014 IEEE Symposium on Security and Privacy (pp. 227-242). IEEE. Gras, B., Razavi, K., Bosman, E., Bos, H., & Giuffrida, C. (2017, February). ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS (Vol. 17, p. 26). Hund, R., Willems, C., & Holz, T. (2013, May). Practical timing side channel attacks against kernel space ASLR. In 2013 IEEE Symposium on Security and Privacy (pp. 191-205). IEEE.

Wu, J. (2020). *Cyberspace mimic defense*. Springer International Publishing. doi:10.1007/978-3-030-29844-9

Xu, M., Lu, K., Kim, T., & Lee, W. (2017). Bunshin: compositing security mechanisms through diversification. In 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17) (pp. 271-283).

Zhang, J. X., Pang, J. M., & Zhang, Z. (2020). Quantification method for heterogeneity on Web server with mimic construction. *Journal of Software*, *31*(2), 564–577.

Zhang, J. X., Pang, J. M., Zhang, Z., Tai, M., Zhang, H., & Nie, G. L. (2019). The Executors Scheduling Algorithm for the Web Server with Mimic Construction. *Computer Engineering*, *45*(08), 14–21.

ENDNOTES

1 https://github.com/obfuscator-llvm/obfuscator/wiki/
2 https://github.com/HikariObfuscator/Hikari/wiki/Usage
3 https://github.com/GoSSIP-SJTU/Armariris
4 https://tigress.wtf/
5 https://github.com/securesystemslab/multicompiler
6 https://www.guardsquare.com/en/products/proguard
7 https://pyarmor.readthedocs.io/zh/latest/usage.html
8 https://obfuscator.io/
9 https://github.com/pk-fr/yakpro-po
10 https://github.com/Zaczero/SharpLoade

Jianmin Pang is a professor in National Digital Switching System Engineering Technological R&D Center. His research interests include high performance computing, cyber security, and quantum computing.

Zhang Fan was born in Wuhu, Anhui, China, on September 24, 1981. He was Ph. D., associate researcher, doctoral supervisor. His main research areas are information communication network and high efficiency computer architecture. He presided over 1 projects on the National Natural Science Foundation, 2 provincial ministerial projects and Participated 6 national level projects and 9 provincial and ministerial projects. He applied for 15 national technical invention patents and, 8 rights were authorized. He published 15 thesis.7 thesis had been retrieves. He written 6 books or textbooks. He was granted one of the First-level Prize of provincial ministerial Hi-tech Progress, one of the Second-level Prize of provincial ministerial Hi-tech Progress and Special-level Prize of Zhengzhou Hi-tech Progress.

Shunbin Li received the PhD degree in information and communication engineering from Zhejiang University, Hangzhou, China, in 2018. He is currently working as an associate research fellow in Zhejiang laboratory. His research interests include VLSI circuits, security, and reconfigurable computing.