LateBA: Latent Backdoor Attacks on Deep Bug Search via Infrequent Execution Codes

Xiaoyu Yi
yxy1234@sjtu.edu.cn
The School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, and Shanghai Key Laboratory of Integrated Administration Technologies for Information Security
Shanghai, Shanghai, China

Gaolei Li
The School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, and Shanghai Key Laboratory of Integrated Administration Technologies for Information Security
Shanghai, China
gaolei_li@sjtu.edu.cn

Wenkai Huang
The School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, and Shanghai Key Laboratory of Integrated Administration Technologies for Information Security
Shanghai, China
sjtuhwk@sjtu.edu.cn

Jianhua Li
The School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, and Shanghai Key Laboratory of Integrated Administration Technologies for Information Security
Shanghai, China
lijh888@sjtu.edu.cn

Xi Lin
The School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, and Shanghai Key Laboratory of Integrated Administration Technologies for Information Security
Shanghai, China
linxi234@sjtu.edu.cn

Yuchen Liu
The Department of Computer Science, North Carolina State University
Raleigh, USA
yuchen.liu@ncsu.edu

ABSTRACT

Backdoor attacks can mislead deep bug search models by exploring model-sensitive assembly code, which can change alerts to benign results and cause buggy binaries to enter production environments. But assembly instructions have strict constraints and dependencies, and these additional model-sensitive assembly codes destroy semantics and syntax and are easily detected by dynamic analysis or context-based detection. To escape from the dynamic analysis-based detection, we propose a novel latent backdoor attack (LateBA) scheme based on the locality principle of program execution, which only poisons a few of infrequent execution codes, minimizing the effects on the original code logic. In LateBA, a progressive seed mutating strategy is designated to change the American Fuzzy Lop (AFL)-based path search tool to pay more attention to infrequent execution codes. With this strategy, the optimal range to positions in the whole program is determined. Subsequently, triggers are target model-sensitive assembly instructions, and try to minimize the variables that have been called in the context instructions in the trigger. Finally, we employ code semantic feature comparisons to select precise trigger injection positions within these ranges. The selection criteria of the trigger injection position is whether the corresponding code segments in this position have a data dependency relationship with other code segments. We evaluate the performance of LateBA over 7 deep bug search tasks. The results demonstrate the attack success rate of the proposed LateBA is considerable and competitive against the baselines.

KEYWORDS

Deep Bug Search, Backdoor Attack, Infrequent Execution Code, American Fuzzy Lop

ACM Reference Format:
Xiaoyu Yi, Gaolei Li, Wenkai Huang, Jianhua Li, Xi Lin, and Yuchen Liu. 2024. LateBA: Latent Backdoor Attacks on Deep Bug Search via Infrequent Execution Codes. In *15th Asia-Pacific Symposium on Internetware (Internetware 2024), July 24–26, 2024, Macau, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3671016.3674806

Internetware 2024, July 24–26, 2024, Macau, China

ACM ISBN 979-8-4007-0705-6/24/07
https://doi.org/10.1145/3671016.3674806

1 INTRODUCTION

Bug search aims to identify defective and non-faulty binary files using limited testing resources. There have been studies on source code-based bug search that provide theoretical insights [3, 25, 32], and the real-world bug search scenarios primarily rely on binary static analysis-based strategies [19, 31]. This is because binary static analysis allows for thorough auditing of the entire target binary files, effectively meeting the user's security requirements. However, these strategies often come with high computational costs, such as assembly code auditing and malicious code feature statistics. Deep learning models have proven effective in understanding semantic features in natural language processing, leading to their widespread application in bug code search and binary code analysis. For instance, research areas such as binary code search [33], code similarity detection [14], vulnerability detection [39] have explored the application of deep learning in learning semantic representations of assembly code. Leveraging deep learning-based strategies

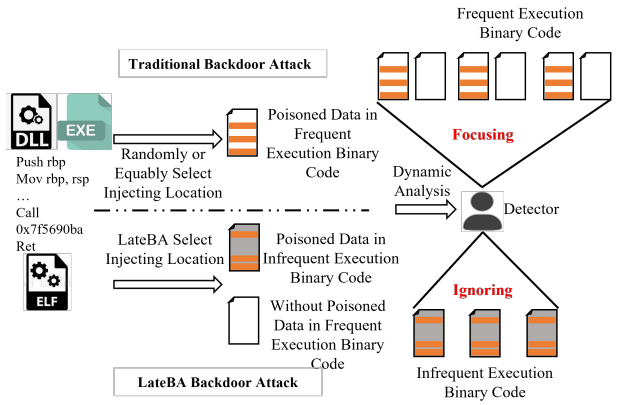

Figure 1: Backdoor attacks against bug search models without considering trigger injection locations are easily found by dynamic analysis, due to the frequent execution codes of binary files are widely distributed, and are prominent parts for dynamic analysis. When the poisoning codes are injected into the location of frequent execution codes, the poisoned codes with incomplete semantics or invalid syntax make victim binary files generate a crash, which reduces the concealment of backdoor attacks.

enables efficient learning of static analysis results and facilitates the search for specific semantic features in code. Consequently, this approach has become the primary method for bug search, replacing the laborious manual exploration of vulnerabilities that relied solely on expert experience.

Recent studies have drawn inspiration from backdoor attacks on deep learning models [20, 28], which provide evidence of vulnerabilities in clean models. Among these attacks, data poisoning attacks that only inject poisoned data called triggers to impact the inference process of the model have emerged as a more covert threat model compared to other backdoor attack methods [9], such as attacking model code or manipulating the training schedule. Taking the bug search model as the attack target is different from the common data poisoning attacks on image classification. To improve the concealment of data poisoning attacks against the bug search model, there are two parts that need to be considered: 1) Stealthily perturbing the inference process of the bug search model. 2) Protecting the normal executing process of poisoned binary files under various dynamic analysis scenarios, which is a unique and important challenge.

Several studies that try to consider data poisoning attacks from option 2) are introduced bellow. PELICAN [41] proposes a poison code strategy that protects the integrality semantic to improve the concealment of data poisoning attacks. PELICAN aims to preserve the coherent program semantics of most parts of the binary code after injecting the poisoning instructions. This is achieved through a randomized micro-execution technique and a solving-based synthesis technique. However, as shown in Fig. 1, these methods only protect the integrality of semantics and can not ensure that all triggers do not lead to program crashes under various dynamic analysis tests. Additionally, reproducing realistic threat scenarios to prevent the program from crashing becomes more challenging due to the substantial computing resources required.

To address this problem, we propose a new potential backdoor attack scheme called LateBA based on the principle of program execution locality. This scheme does not require complex micro-execution and symbolic execution methods to improve the concealment of data poisoning attacks. The LateBA consists of two steps: 1) Exploring hidden injection location: since the infrequently executed code has limited relationship with context, it is the optimal injecting location of the trigger. 2) Find the accurate location for injecting poisoned code and generating a more efficient trigger: once the infrequently executed code segments are identified, determining the accurate and secluded injecting location within the code segments is the next task. To prevent the influence diffusion brought by executing triggers with incomplete semantics, LateBA selectively deletes certain instruction sequences within the infrequently executed codes that have more data dependency relationships with the context.

The main contributions of our work are summarized as follows:

- We propose a novel latent backdoor attack scheme termed LateBA against the bug search model. Different from other backdoor attack strategies, LateBA utilizes infrequent execution codes to prevent dynamic analysis-based backdoor detection.
- We designed a progressive seed mutating strategy designated to change the American Fuzzy Lop (AFL)-based path search tool to pay more attention to infrequent execution codes.
- We propose an evaluation method to demonstrate that the proposed method is effective against dynamic analysis and improves the efficiency of backdoor attacks. LateBA significantly reduces the number of triggers executed during dynamic analysis by at least 75%, and it also effectively reduces the number of crashes caused by executing triggers by 85%.

2 THREAT MODEL

Our goal is to inject backdoors into the model via data poisoning. This is needed to find infrequent execution code as the injection location to avoid dynamic analysis. We focus on transformer models used in binary code analysis, which are primitives for a variety of cybersecurity applications: vulnerability hunting, malware analysis, decompilation, and forensic analysis. Transformer was the most effective model in these analyses, outperforming other models such as CNN, RNN, and LSTM. Therein, the attack models used in other applications (e.g. network traffic-based intrusion detection) require completely different trigger injection techniques (in order to preserve traffic semantics). We therefore consider it beyond the scope of this article.

In the real software security detection scene, manipulating the model is extremely difficult to achieve. We only consider black-box scenarios. In this scenario, the attacker does not have access to the object model. Therefore, we exploit the locally executing principle of these executable files. It means that the program shows locality when executed, that is, within a period of time, the execution of the entire program is limited to a certain part of the program, e.g., the modifying dependency library address behavior or the deleting data behavior. Therefore, an attacker can inject a backdoor

trigger on a piece of code that he has access to, and then use it to inject a location that is rarely executed by users, thus reducing the probability of the trigger being discovered during dynamic analysis. Additionally, black-box attacks utilizing gradient approximation can also be exploited. We leave this to our future work.

3 RELATED WORK

3.1 Bug Search Model

To provide a steady and secure production environment when updating software systems, many studies focus on deep learning-based bug search methods. Murali et al. proposed a Bayesian method that generates a probabilistic model of API calls, which improved the robustness of the bug search model in an unstructured code corpus[16]. Bin et al. utilized normal specifications to compare to unknown code, which can find unusual and possibly error code [11]. But the error code cannot be generated in bulk, the DeepBugs proposed a finding bug model that only needs to learn from positive samples, and it can reduce the time to build the sample set [19]. Meanwhile, Polisetty et al. used a convolution neural network and logistic model to examine the effectiveness of bug location [18].

3.2 Fuzz Testing

Fuzz testing was proposed by Miller et al. [15]. It was mainly used to test the robustness of UNIX programs. In 1999, it was extended to include security testing. During this period, black-box fuzz testing has been widely used, including well-known fuzz testing tools such as SPIKE [4]. Blackbox fuzzing randomly generates test cases and has a fast testing speed. However, it lacks access to program internal information, which limits the full exploration of deep program logic. In 2008, Godefroid et al. [5] developed SAGE, a white-box fuzzer that combines symbolic execution and fuzz testing techniques to generate test cases. Compared with black-box fuzz testing, white-box fuzz testing can use program internal information to generate test cases related to specific paths. Nevertheless, the complexity of software and the limitations of solvers [1] pose obstacles to the effectiveness of fuzz testing, namely, thorough testing within a limited time frame. Therefore, researchers have shown great interest in how to strike a balance between the utilization of program internal information and testing efficiency. This has promoted the development of greybox fuzz testing. At the end of 2013, Zalewski released a greybox fuzzer American Fuzzy Lop (AFL). During the fuzz testing process, AFL uses instrumentation to collect path information of the target program and uses coverage to guide the generation of test cases, namely coverage-based greybox fuzzing (CGF) [22].

3.3 Backdoor Attacks in Binary Analysis

A backdoor attack is a training phase attack. The attacker installs a backdoor trigger into the target model in some way before the start of training or during the training process, so that the prediction results of the model can be accurately controlled during the testing phase [10, 27, 29, 37]. With the popularity of Machine Learning as a Service (MLaaS) and Model as a Service (MaaS) and the reliance on network data for training large models. The goals of the backdoor attack are: (1) The backdoor model has normal accuracy on clean test samples; (2) If and only if the test sample contains a preset backdoor trigger, the backdoor model will be generated by the attacker Prespecified prediction results. Among them, goal (1) ensures the concealment of backdoor attacks, and goal (2) ensures that the backdoor model can be arbitrarily manipulated by attackers.

Researchers have demonstrated such attacks on image recognition[21], natural language processing[40], reinforcement learning [30], etc. Backdoors can be injected into these systems via model poisoning [12] and neuron hijacking [13]. LateBA belongs to the last category, where the adversary only has access to pre-trained models with a small number of samples. Existing natural backdoors mainly target the field of computer vision, and the source code analysis[8]. LateBA targets binary code analysis tasks, which need to have an insight into the binary code semantic learning.

As far as we know, most of the existing backdoor attacks on deep bug search models use symbolic execution or micro-execution to improve the concealment of poisoned data[36], and these efforts will inevitably lead to an increase in attack costs (for example, obtaining code information around the injection location, long symbolic execution, etc.). The infrequently executed code injection poisoned data proposed in this paper uses the local execution principle of the program to infer the rarely executed locations in the victim binary program to hide the poisoned data.

4 DESIGN OF LATEBA

4.1 Overview

As shown in Fig .2, the whole scheme of LateBA is divided into two steps: 1) Finding injection ranges, and 2) Finding the accurate location for injecting poisoned code and generating triggers in binary files. In the 1) step, the given binary files are used to search for the seed set and the corresponding number of branches by determining the mutation strategy at parts (a) and (b), as detailed in Section 4.2.1. According to the above-captured information, we utilize a random mutation strategy to find the infrequently executed basic block at part (c)(d) as the infrequent execution ranges in Section 4.2.2. In the 2) step, we utilize the code semantic comparison strategy to find the instruction sequences that can impact the prediction of the bug search model at part (e). In part (f), we delete some instruction sequences that have data dependencies with context and generate a trigger in victim binary files in Section 4.3.

4.2 Finding Injection Ranges

4.2.1 Deterministic Mutation for Initializing Seed Set. The deterministic mutation utilizes the signal of finding a new basic block to evaluate the optimal seed input and the optimal byte in seed inputs. For unknown binary files, the reasonable input seed set is required by the random mutation for searching infrequent execution basic block. Meanwhile, since these basic blocks are connected by call branches that must relate to two basic blocks, the number of call branch hits can help the analysis process if the input seed has the optimal power for exploring infrequent execution basic blocks. Specifically, the searcher that aims to find the infrequent code of binary files selects the lower number of call branch hits that can work out some advantage input parameters and the bytes in the input parameter for later deeply infrequent execution code searching. So we update the whole hit branch record when the basic block of binary code passes a call branch. But it is an unsuitable

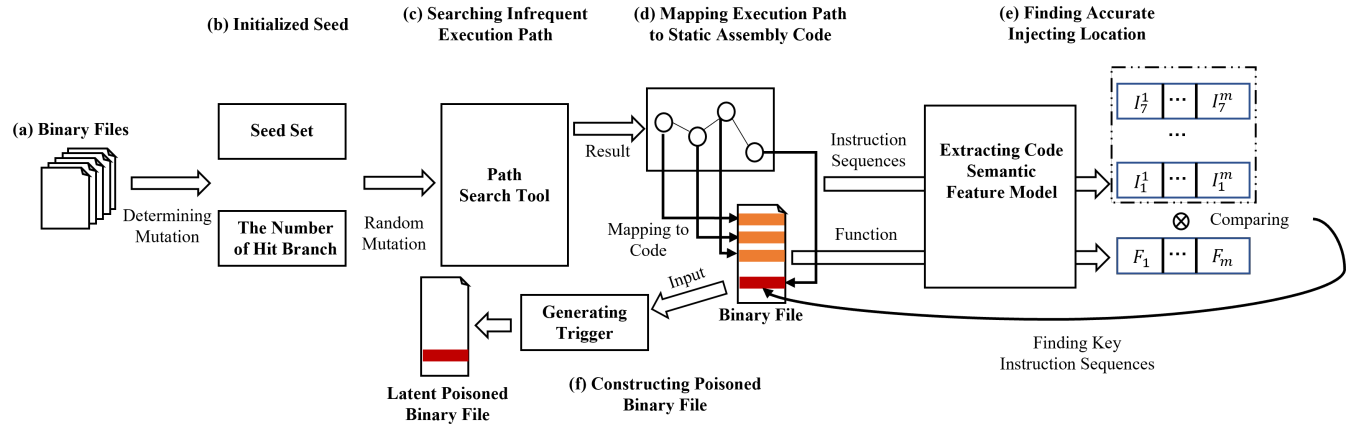

Figure 2: The overview of our latent backdoor attack framework LateBA is divided into two parts: 1) Finding infrequent execution code as injection ranges of poisoned data, and 2) Finding the accurate location for injecting poisoned code and generating triggers in binary files. The colorful parts are the infrequent execution code segment by the random mutation strategy generating. Therein, the red is the infrequent execution code segment that can impact the prediction of the bug search model.

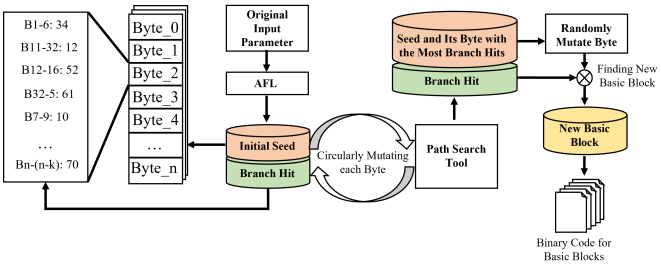

Figure 3: The description of finding the covert location for injecting poisoned code. The left of "Circularly mutating each byte" is the deterministic mutation, and the right is the random mutation.

way to use the bytes of seed input that is selected by exploring new execution paths strategies to achieve deterministic mutation, because these branches with few hits do not include the infrequent execution code, like the precursor and the successor of this branch have been executed with 5000 times, but the current branch only has been executed 10 times.

Based on this idea, the LateBA proposes a selection method that determines the mutation byte in the input seed to achieve the deterministic mutation process. Therefore, we download some common input seeds. Then we randomly generate one-byte data and use this to cover each byte of the original seed, which is circularly executed and can record the new branches that are generated. After this, we choose the covered byte that causes the maximum number of new basic blocks to be executed as the optimal byte at the random mutation stage. The detailed rules are shown as follows, *rule* shows the rule of mutation, *deterministic_mutate* is the deterministic mutate method, *mutate* is the single mutate process, thereinto the S_{behave} is the type of mutation that includes: 1) O: Overwriting b byte(s), 2) I: Inserting b byte(s), 3)D: Deleting b byte(s), the b describes the number of bytes that are mutated, the $Input$ is input seed and the $Input_i$ shows the begin location of mutation.

$$rule(behave, b) = \{behave \in \{O, I, D\}, mutate(S_{behave}, b)\} \quad (1)$$

$$deter\_mutate = \{i = [1, |I|), mutate(Input_i, rule(O, 1))\} \quad (2)$$

Based on the above result, in order to initialize a seed set that is more likely to generate the infrequent execution code, we construct a dynamic selection rule to reply to different binary files. The continue call branches with fixed range are identified target branches, and the input seeds that can lead to the target branch are updated in an initial seed. The definition of fixed range is shown as follows, B_{cur} is the branch set that includes all covered branches, $branch\_Hit(b)$ is the number of branch hits of b, the fixed range can dynamically change, and all the branches that satisfy fixed range are added into initializing seed set.

$$B_{cur} = \{b \in B : branch\_Hit(b) > 0\} \quad (3)$$

$$min(B_{cur}) < fix\_range < 2 * min(B_{cur}) \quad (4)$$

4.2.2 Random Mutation for Finding Infrequent Execution Code Segments. After initializing the seed set and the number of branch hits, we mutate the input seed in certain mutation methods and explore many new basic blocks with a lower number of branch hits. In these new basic blocks, the consecutive basic blocks are defined as infrequent execution basic blocks. We utilize the random mutation strategy to mutate the optimal byte and other bytes around this byte, which can find some consecutive infrequent execution basic blocks that correspond to binary code as the infrequent execution binary code.

For the random mutation stage, we think that some bytes that are prioritized to mutate can explore many infrequent execution binary codes. Specifically, one-byte changes or more changes centered

around these bytes in seed input can execute more the infrequent execution code, because the input parameters of binary files are structural data, and if one byte causes the execution of infrequent executed binary code, mutating others bytes around this byte can find other infrequent execution binary code in more possibility. Based on the result of deterministic mutation, we utilize the random mutation type and the random location of the mutation. The available range of random location mutation is selected in seed input by deterministic mutation. Moreover, after the deterministic mutation method selects some optimal bytes, these bytes that have a higher possibility of finding infrequent execution code become the start position. Then we set 4 bytes as a stride and randomly generate the number of strides as offset based on the start position, which is mutated bytes. The detailed information is as follows, the *base_byte* can touch more unexecuted basic blocks by deterministic mutation, the j is the number of strides, the *mutate_bytes* is the bytes of input seed that is mutated by random mutation strategy, and the *random_mutate* is the random mutation process.

$$mutate\_bytes = base\_byte + j * 4 \in seed\_input \tag{5}$$

$$\begin{aligned} random\_muate = \{ & j = [1, k), mutate(I_{base\_byte}, \\ & rule(mutate\_type, j * 4))\} \end{aligned} \tag{6}$$

In addition, we record the offset of each new basic block on the function's start address, which can obtain the infrequent execution binary code according to the new basic blocks that are found by random mutation. However, this binary code has a wide range of binary files, requiring the injection of numerous test-poisoning samples to find the optimal location.

4.3 Generating and Injecting Data Dependency-agnostic Triggers

After we determine the range of injecting triggers that can improve the high concealment of backdoor attacks, it is next extremely important to build effective triggers. These triggers can use fewer poisoned samples to achieve a successful backdoor attack. For this task, since checking the code feature of the core logic of healthy binary is the prime rule of most bug search models, as shown in 3.1, the LateBA utilizes the code semantic feature to find core code and injects poisoned code into the code with core logic to misclassify the prediction result of bug search model.

For finding core code, as the minimum logic unit of the program, the function can support a complete logic process of computing data. Meanwhile, in the target function with the infrequent execution code, there is one code fragment or several code fragments that can achieve this logic process of the target function. These code fragments are called core code in the target function. We extract the semantic feature of the core code of the target function and find the semantic feature that is similar to the semantic feature of the target function as core code. On the basis of semantic extraction, we averagely split each function into several instruction fragments. Since the basic block has only a single entry and a single exit, it is easy to sort out the dependencies between the assembly instruction sequences. We divide the function according to the boundaries of basic blocks, and set the number of basic blocks with the most basic blocks in a single function as the fixed length of the input. To prevent the execution order from being broken, we divide the function into sequences according to the call and ret instructions. The semantic features of code (function and its instruction sequence) are extracted by the pre-training model and the semantic learning model [38]. Specifically, the semantic features within instructions utilize embedding instructions to capture the feature vector of instruction. Then, the LSTM model cyclically learns these concatenated feature vectors for extracting the feature of the instruction sequence, like function, and code fragment. Finally, we compare the cosine distance of feature vectors between instruction sequences and corresponding functions. The one most similar to the feature vector of the function is the optimal backdoor injection location.

For generating data dependency-agnostic triggers, since the contract execution process depends on many third-party libraries, modifying or adding instructions must maintain correct data dependencies to increase the concealment of backdoor injection. To ensure syntactically correct assembly encoding after inserting instructions, we limit the modification and addition of instructions that have data dependencies with the function context. Since a lot of high-level language information (variable names, function names, etc.) is lost during the compilation process, this high-level language information is redirected into registers or memory addresses. This article can only search for registers that have no value transfer relationship with the recently called register to build trigger instructions. When fetching the target instruction fragment, the flip-flop generator avoids using operands that are registers holding the running state, such as RBP, RSP, etc. Instead, it uses immediate numbers as operands and swaps between infrequently used registers as opcodes to inject these instructions as triggers. Through experimentation, we find that this approach can achieve classification results that misguide the model with only a few instructions. Finally, we find that the instruction fragment characteristics in the target function are similar to the target function characteristics, and it is used as the optimal poisoned data injection location.

5 EVALUATION

We implemented our attack that is conducted on a dedicated server with a Xeon(R) 8259CL $CPU@2.50GHz \times 16$, one GTX 3090 GPU, 12GB memory, and 1 TB SSD. We execute the path search algorithm in the parallel model, which helps the evaluation to take advantage of the multi-core. We designed the evaluation method by focusing on two targets including searching infrequent execution code and generating a trigger of a backdoor attack. For searching the infrequent execution code, since the code coverage rate is the evaluation of finding new paths rather than searching for the infrequent execution code, we establish a new evaluation rule specifically designed for identifying the infrequent execution code. Meanwhile, we take four popular software (supporting multiple operating systems) and different levels of optimization (O1, O2, O3) as data, which can test the adaptive capacity and the robustness of our tool. Therein, the data for each compilation optimization level includes different instruction architectures, ARM, MIPS, and X86, thereby expanding the scope of experimental targets to verify the feasibility of LateBA. At the same time, we try to use virtualized low-computing resource

devices, whose total computing resources can only reach 1/3 of the existing ones. Stable experimental results are captured after 72 hours of testing each software with the infrequent execution code searching tool. For generating the trigger of a backdoor attack, in each software, we utilize the standard cross-validation method to evaluate the effectiveness of the backdoor attack, which splits the datasets into 1/3 for training, 1/3 for testing, and 1/3 for validation.

5.1 Datasets

We select four mainstream tool libraries (in the latest versions when tested) to make a benchmark dataset for our model and promote the evaluation result generalizable, including the Openssl is a widely used library of cryptographic tools in the network socket layer and supports cryptographic services in other domains. The Curl is a packet transmitter and receiver that supports a variety of network layer protocols. The Busybox is also an open source code library that includes the most common Linux commands, which can achieve these functions with minimal codes. They are all binary files compiled on ARM, MIPS, and X86 platforms with different optimization levels ($O1$, $O2$, and $O3$). The whole dataset consists of 9,100 executable paths, 120 million functions, and 6.1 billion instructions in total.

Therein, the size of input files vastly impacts the efficiency of initializing seed sets, like 10KB files as input files and 5KB input files may more cost than 1000ms or 2000ms. Selecting the minimum size seed input into the seed set can extremely improve the efficiency of initializing seed sets. After minimizing the original input file, we perform this initialization seed sets strategy for each software, which executes 24 hours one time.

5.2 Latent Backdoor Attack

In our empirical study, we want to evaluate whether the infrequent execution code is a kind of optimal injection location to improve the invisibility of backdoor attacks, and the effectiveness of backdoor attacks that are constructed by the LateBA. To clearly evaluate the performance of LateBA, we conduct experiments to answer the four questions as follows:

- Q1: Can our designed infrequent execution code searching tool in LateBA precisely find infrequently-executed codes?
- Q2: Whether utilizing infrequent execution code as trigger injection locations can really improve the concealment of backdoor attack?
- Q3: Does the data dependency-agnostic trigger optimization strategy actually increase the attack efficiency under few poisoned samples scenarios?
- Q4: Can our proposed LateBA adapt to different compilation optimization levels?

In Q1, we describe that the searching infrequent execution code tool can find some hardly executed code in target software when the target software is in a state of normal execution, and Q2 explains why the infrequent execution code can improve the invisibility of backdoor attack in bug search model, and show the real performance when utilizing this location to place poisoning code. In Q3, for different poisoning rates, we display the optimizing trigger method which is the threat model with more threat capability. The Q4 considers that the performance of LateBA combats other interference factors in the real scene of software systems.

5.2.1 Question1. The location of infrequent execution code is a prerequisite for injecting poisoning code. For this question, the optimal evaluation strategy is the statistical results of touched code, and the scenario is the normal execution of binary files. Although the common coverage-based path search tool often uses touching new branches as positive behavior to construct incentive mechanisms, it focuses on finding new basic blocks, which can not meet the requirements of evaluation. For example, turning the unconditional jump to the conditional jump only changes one basic block in the same path, which may be identified as new code. Since this task of finding infrequent execution code mainly relies on the mutation of seed input, we utilize the target code mutation rate to evaluate the availability of rare search tool.

In Fig. 4, each tool library is executed 24 hours by the AFL path search and the infrequent execution code searching tool in LateBA. There are 100 input files for each search tool, and these input files averagely divided into 4 parts. Each part only executes one kind of data process in the target tool library. Specifically, as shown in Fig. 4, since the OpenSSL is a cryptogram library that provides implementations of some basic cryptographic algorithms, many mutated input seeds can not satisfy the requirements of some specific cryptographic algorithms, which interrupt the execution process. It is shown in a) figure, the result of evaluating OpenSSL can not touch a higher proportion in target function than other tool library. Then, these input files are mutated to new input files for executing unknown code, we check how many of these codes belong to the objective function. For this process, we update the record of the hit basic blocks within the target function to count the executed code proportion in the target function and take snapshots every 2 hours. We run the tested tool with each input file for 24 hours. Although we want to show the performance of our infrequent execution code searching tool, time is the most precious resource for most attackers. The inputs of Curl are some short commands and network packets, which can impact executed functions by mutating bytes. In the b) figure, the code proportion of the target function fleetly increases by 18-th hour. Finally, we calculate the average of the results obtained for 100 input files and generate these figures. In Fig. 4, our designed infrequent execution code search tool has better performance than other path search tools when searching infrequent execution code.

5.2.2 Question2. After searching infrequent execution code, The next step is to evaluate whether infrequently executed code improves the concealment of backdoor attacks for the bug search model. However, the LateBA's prime principle different from the other enhancing concealment methods, which makes the concealment evaluation of LateBA is different from the common evaluation methods. Specifically, the prime goal of LateBA is to ensure that the binary files can execute properly with the most of input parameters, after injecting triggers from the LateBA. Therefore, our evaluation method is to test the availability of binary files injected with a backdoor attack trigger.

To test the availability of poisoned binary files, we use the ablation study to verify the concealment of other backdoor attacks without the help of an infrequent execution code search tool. At first,

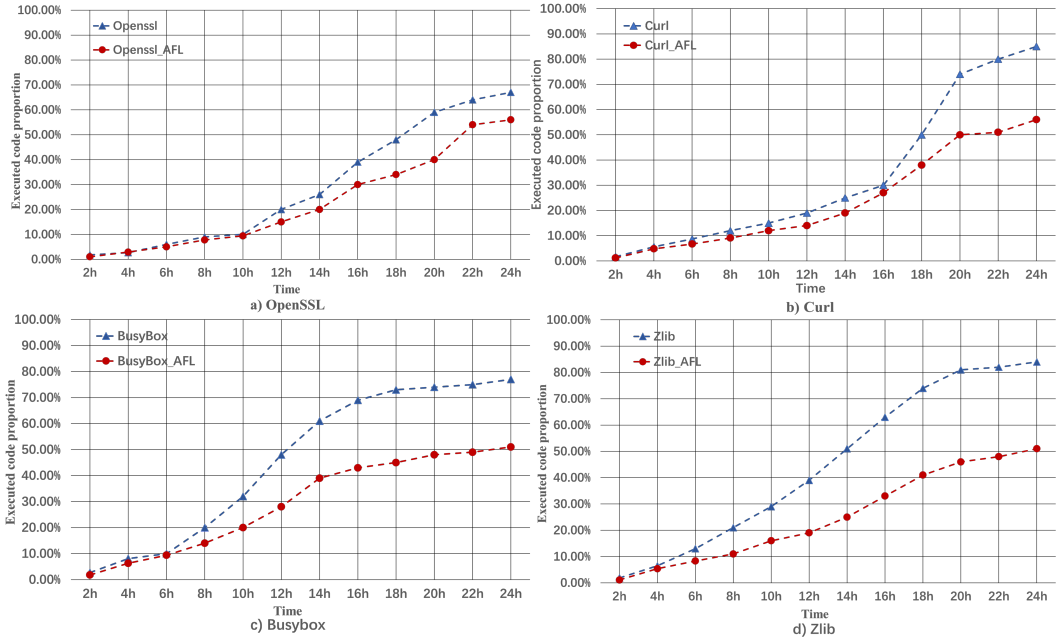

Figure 4: The Depiction of the infrequent code explores the process by calculating the executed code proportion in the target function. The AFL-based fuzzing tool's execution result is taken as the baseline for comparison. The red line is the result of the AFL-based fuzzing tool, and the blue line is the result of LateBA.

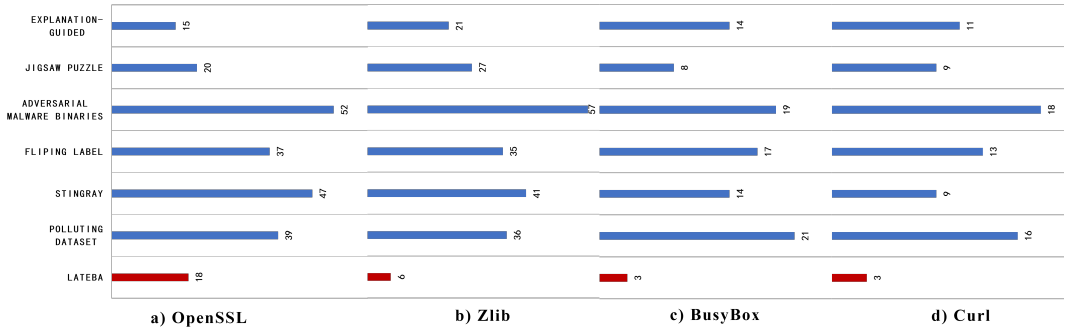

Figure 5: Description of the global program crash by mutation input seed and the poisoning code at the same time. The crash number of binary files that are poisoned by LateBA compares with the crash number of poisoned binary files by the other backdoor attack respectively. The red part is the result of LateBA.

we utilize various backdoor attack strategies to generate triggers and inject target tool libraries, including OpenSSL, Curl, BusyBox, and Zlib. Meanwhile, these injected triggers are labeled with corresponding signs. Then, we use the same input seed set and same seed mutation strategy to run these tool libraries and record some information about the number of program crashes, the number of executed trigger signs, and the number of program crashes caused by injected triggers. This information allows us to determine which trigger of a backdoor attack has higher concealment.

For the baseline of the ablation study, to the best of our knowledge, There are few existing publications on backdoor attack targeting bug search, whereas malicious software detection is the most researched area. We utilize 6 backdoor attack, include Explanation-guided backdoor attack[24], Jigsaw Puzzle [35], Adversarial Malware Binaries [6], Fliping label-based backdoor attack [7], StingRay [26], and Polluting training dataset-based backdoor attack[2]. Thereinto, the Explanation-guided backdoor attack[23] and the Jigsaw Puzzle [34] provide source code implementation, and the experimental data for the other methods come from their papers. At first,

Table 1: Attack effectiveness of LateBA with different compiler optimization level. *ASR* denotes the attack success rate. *ACC_model* denotes the accuracy of deep bug search model after injecting poisoned data. *LateBA_O,LateBA_C,LateBA_B* and *LateBA_Z* denote the input dataset is Openssl, Curl, Busybox and Zlib respectively.

Optimization level	Target	ASR	ACC model	Optimization level	Target	ASR	ACC model	Optimization level	Target	ASR	ACC model
O3	Explanation-guided	0.452	0.612	O2	Explanation-guided	0.775	0.675	O1	Explanation-guided	0.891	0.748
	Jigsaw Puzzle	0.553	0.681		Jigsaw Puzzle	0.844	0.720		Jigsaw Puzzle	0.909	0.772
	Adversarial Malware Binaries	0.428	0.429		Adversarial Malware Binaries	0.682	0.572		Adversarial Malware Binaries	0.837	0.629
	Flipping label	0.426	0.582		Flipping label	0.737	0.691		Flipping label	0.836	0.721
	Polluting training dataset	0.593	0.620		Polluting training dataset	0.898	0.767		Polluting training dataset	0.914	0.801
	LateBA_O	0.741	0.729		**LateBA_O**	0.84	0.814		**LateBA_O**	0.919	0.882
	LateBA_C	0.741	0.692		**LateBA_C**	0.84	0.786		**LateBA_C**	0.919	0.802
	LateBA_B	0.741	0.729		**LateBA_B**	0.84	0.841		**LateBA_B**	0.919	0.901
	LateBA_Z	0.741	0.641		**LateBA_Z**	0.84	0.724		**LateBA_Z**	0.919	0.894

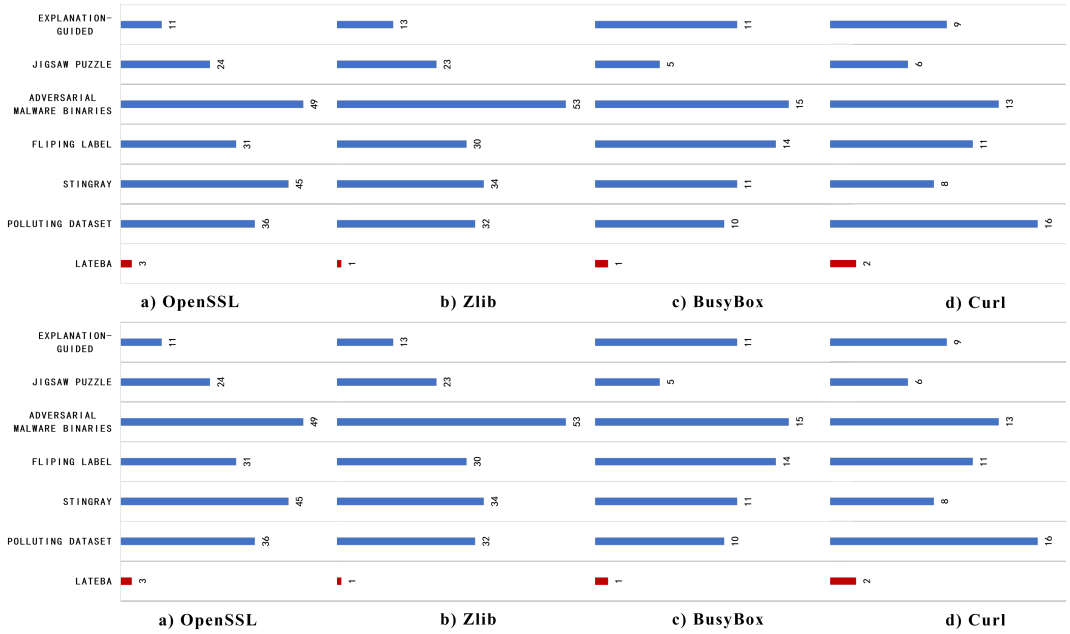

Figure 6: The upper part show that the number of triggers are executed by AFL-based path search tool. And the half bottom show that the number of program crash is caused by the trigger. The red part is the LateBA's result. And the blue parts are the others triggers' result.

these backdoor attack tools construct triggers in all binary tool libraries, and instrument a counter called $trigger\_i$ at the poisoned position. For example, if the poisoning target is an instruction, the sign is labeled at each poisoned instruction. The i presents the identification of the trigger. The $trigger_i = 1$ represents that the i trigger is executed by AFL-based fuzzing. Meanwhile, these library tools that are injected with poisoned instructions are turned into new binary files by rewrite. Then, We prepare 100 input seeds as input seed set for these new binary files, and when the execution state of the new binary files is stabilized (no more new path), we collect all the counters and compute their mean, as shown in Fig. 5 and Fig. 6. Fig. 5 shows the total number of program crashes, and Fig. 6 shows the number of executed triggers and the number of program crashes that are caused by executing triggers in the evaluation process. For the number of program crashes, excluding the number of crashes caused by triggers, the crashes number of programs that are poisoned by LateBA is basically equal to the number of crashes that are obtained by fuzzing the normal tool library under the same fuzzing conditions. In Fig. 6, it shows the triggers that are generated by LateBA are hardly found by many times dynamic analysis than the other triggers of a backdoor attack.

The above result shows the LateBA can improve the concealment of backdoor attacks.

For the above evaluation results, the concealment achieved by LateBA allows attackers to consider as little as possible how the poisoned data should ensure the integrity of the context dependency. Since the compilation process loses a lot of logical information, the only available logical rationality check for the binary file is the context dependency. Infrequently executed code is rarely called by the context, which improves the concealment of the poisoned data, thereby enabling the attacker to implement a backdoor attack on the bug search model without consuming a large amount of attack cost, such as trigger construction time and computing resources, attack penetration times, etc.

5.2.3 Question3. Besides determining the location of injection poisoning code, the other task of LateBA that improves the efficiency of backdoor attacks is generating data dependency-agnostic triggers. In this task, the data dependency-agnostic trigger meets two goals that include two goals: first, injecting into core code locations that can significantly impact the prediction behavior of the bug search model, and second, minimizing the data dependency of these codes with the context. For the first goal, the LateBA poisons core code to the misclassification bug search model. For the second goal, the less data dependency of trigger can reduce the program crash number that is generated by changing core code. Although the infrequent execution code as the injection location has been high concealment for the dynamic analysis at the binary level, the two goals of the data dependency-agnostic trigger are designed against the more complex backdoor attack scenarios. The lower the poisoning rate, the harder to leave a trace of the attack, which offers many chances in different threat models. To evaluate the performance of the LateBA against complex attack scenarios, we split the poisoning rate into some small steps in the lower poisoning rate range, and compare their attack success rate for checking this performance.

Table 2: Comparing the execution state between different poisoning rate, *ASR* is the attack success rate, *Firstsuccesstime* is the time cost of first attack success which is the number of passes through the trigger, and *Poisoned instruction number* is the number of poisoned instruction.

Poisoning rate	ASR	First success time	Poisoned instruction number
0.5%	81.71%	280-th trigger	1.050E+07
1.0%	82.15%	187-th trigger	2.102E+07
1.5%	83.49%	103-th trigger	3.159E+07
2.0%	83.93%	75-th-trgger	4.201E+07
2.5%	85.76%	13-th trigger	5.251E+07
3.0%	87.02%	3-th trigger	6.321E+07
3.5%	89.59%	2-th trigger	7.352E+07
4.0%	88.08%	4-th trigger	8.471E+07
4.5%	89.91%	1-th trigger	9.452E+07
5.0%	92.18%	2-th trigger	1.053E+08

In the lower poisoning range ($\leq$ 5.0%), we set the one step of poisoning rate as 0.5% and the start point as 0.5%. We record the attack success rate (ASR), the time cost of first attack success, and the number of poisoned instruction, where each variable is the average of testing four tool libraries. Thereinto, the time cost of first attack success is represented by the identification number of trigger, and these identification numbers are recorded in order of execution.

As shown in Table 2, the ASR can touch 80% at very low poisoning rate, like 0.5%. But the first success backdoor attack costs longer time, about passing 280 trigger. And when the poisoning rate is 3%, LateBA's backdoor attack can produce promising results.

5.2.4 Question4. The different binaries are produced with different compilation optimization levels, which is the common scenario of real-world. This scenario can represent different semantic features when using the different compilation optimization levels to compile same source code. But the different semantic features can impact the attack performance of LateBA, which requires stronger robustness to guarantee the basic performance of the backdoor attack strategy.

To assess the robustness of the LateBA, we construct four test datasets that come from four tool libraries compiled with different optimization levels and use the same value of poisoning rate (3%) that can prevent generating unstable ASR. The [17] serves as a target for backdoor attacks. It embeds various useful information that is usually ignored by existing technologies. This information includes the APIs used, API calls, connection types, and key parameters provided to each API. A custom neural network model is designed, which is trained with node-level and semantic-level embeddings to achieve malware detection. We record the recognition accuracy after testing on each dataset. As shown in table 1, the high compilation optimization level affects the performance of backdoor attacks, and the LateBA has a higher ASR than the other backdoor attacks, which shows the LateBA has better performance.

6 CONCLUSION

In this paper, the concealment of backdoor attacks is validated by comparing the crash numbers of poisoned binary files between the proposed LateBA approach and other backdoor attack strategies. The key idea is that LateBA specifically targets the injection of triggers in infrequent execution code, which makes the backdoor attack more concealed. This is validated by monitoring the proportion of executed code during a 24-hour period, where LateBA demonstrates a superior ability to identify infrequent execution code compared to AFL, a popular fuzzing tool. LateBA only poisons a few infrequent execution code segments, minimizing the impact on the original code logic. A progressive seed mutating strategy is designed to modify the AFL-based path search tool, directing it to focus more on infrequent execution codes. This enables the determination of the optimal range to positions. The experimental results demonstrate that the LateBA approach significantly enhances the concealment and stability of backdoor attack detection compared to existing methods.

Individual dynamic analysis strategies, such as taint analysis, dynamic instrumentation, symbolic execution, etc., all require a high code coverage rate to be implemented. The evaluation of program crashes caused by poisoned data in the experiment proved that LateBA can reduce the probability of being discovered by a single dynamic detection strategy by at least 23.27%. For strategies that use other technologies to assist in backdoor attack detection, such

as assembly instruction semantic detection, abstract syntax tree matching, etc., the time to detect poisoned data will be shortened. In this regard, LateBA needs further research.

7 DATA AVAILABILITY

We provide the code and data of our experiments for reproduction at https://github.com/yxy-whu/LateBA.git.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their time and invaluable feedback to improve our study. This work is funded by the National Nature Science Foundation of China under Grant No. 62202303, U20B2048, and U21B2019.

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[2] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

[3] Rudolf Ferenc, Dénes Bán, Tamás Grósz, and Tibor Gyimóthy. 2020. Deep learning in static, metric-based bug prediction. *Array* 6 (2020), 100021.

[4] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[5] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[6] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*. IEEE, 533–537.

[7] Chaoran Li, Xiao Chen, Derui Wang, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2021. Backdoor attack on machine learning based android malware detectors. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (2021), 3357–3370.

[8] Jia Li, Zhuo Li, HuangZhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2023. Poison attack and poison detection on deep source code processing models. *ACM Transactions on Software Engineering and Methodology* (2023).

[9] Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. 2022. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems* (2022).

[10] Yanzhou Li, Shangqing Liu, Kangjie Chen, Xiaofei Xie, Tianwei Zhang, and Yang Liu. 2023. Multi-target backdoor attacks for code pre-trained models. *arXiv preprint arXiv:2306.08350* (2023).

[11] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: mining more bugs by reducing noise interference. In *Proceedings of the 38th International Conference on Software Engineering*. 333–344.

[12] Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. 2020. Composite backdoor attack for deep neural network by mixing existing benign features. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 113–131.

[13] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning attack on neural networks. In *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc.

[14] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*. Springer, 309–329.

[15] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[16] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Finding likely errors with Bayesian specifications. *arXiv preprint arXiv:1703.01370* (2017).

[17] Minh Tu Nguyen, Viet Hung Nguyen, and Nathan Shone. 2024. Using deep graph learning to improve dynamic analysis-based malware detection in PE files. *Journal of Computer Virology and Hacking Techniques* 20, 1 (2024), 153–172.

[18] Sravya Polisetty, Andriy Miranskyy, and Ayşe Başar. 2019. On usefulness of the deep-learning-based bug localization models to practitioners. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 16–25.

[19] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[20] Ge Ren, Jun Wu, Gaolei Li, Shenghong Li, and Mohsen Guizani. 2022. Protecting intellectual property with reliable availability of learning models in ai-based cybersecurity services. *IEEE Transactions on Dependable and Secure Computing* (2022).

[21] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. 2022. Dynamic backdoor attacks against machine learning models. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 703–718.

[22] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[23] Giorgio Severi. 2022. Explanation guided backdoor implementation. Retrieved June 7, 2023 from https://github.com/ClonedOne/MalwareBackdoors

[24] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In *30th USENIX security symposium (USENIX security 21)*. 1487–1504.

[25] S Sivapurnima and D Manjula. 2023. Adaptive Deep Learning Model for Software Bug Detection and Classification. *Computer Systems Science & Engineering* 45, 2 (2023).

[26] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. 2018. When does machine learning {FAIL}? generalized transferability for evasion and poisoning attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 1299–1316.

[27] Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring neural code search. *arXiv preprint arXiv:2305.17506* (2023).

[28] Guanhong Tao, Yingqi Liu, Guangyu Shen, Qiuling Xu, Shengwei An, Zhuo Zhang, and Xiangyu Zhang. 2022. Model orthogonalization: Class distance hardening in neural networks for better security. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1372–1389.

[29] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what i want you to see: poisoning vulnerabilities in neural code search. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1233–1245.

[30] Lun Wang, Zaynah Javed, Xian Wu, Wenbo Guo, Xinyu Xing, and Dawn Song. 2021. Backdoorl: Backdoor attack against competitive reinforcement learning. *arXiv preprint arXiv:2105.00579* (2021).

[31] Hao Wu, Hui Shu, Fei Kang, and Xiaobing Xiong. 2019. BiN: A two-level learning-based bug search for cross-architecture binary. *IEEE Access* 7 (2019), 169548–169564.

[32] Geunseok Yang, Kyeongsic Min, and Byungjeong Lee. 2020. Applying deep learning algorithm to automatic bug localization and repair. In *Proceedings of the 35th Annual ACM symposium on applied computing*. 1634–1641.

[33] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244.

[34] Limin Yang. 2022. Jigsaw Puzzle. Retrieved June 15, 2023 from https://whyisyoung.github.io/JigsawPuzzle

[35] Limin Yang, Zhi Chen, Jacopo Cortellazzi, Feargus Pendlebury, Kevin Tu, Fabio Pierazzi, Lorenzo Cavallaro, and Gang Wang. 2023. Jigsaw puzzle: Selective backdoor attack to subvert malware classifiers. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 719–736.

[36] Wenkai Yang, Yankai Lin, Peng Li, Jie Zhou, and Xu Sun. 2021. Rethinking stealthiness of backdoor attack against nlp models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 5543–5557.

[37] Zhou Yang, Bowen Xu, Jie M Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering* (2024).

[38] Xiaoyu Yi, Gaolei Li, Ao Ding, Yuqing Li, Zheng Yan, and Jianhua Li. 2023. AdvBinSD: Poisoning the Binary Code Similarity Detector via Isolated Instruction Sequences. *IEEE SpaCCS-2023* (2023).

[39] Xiaoyu Yi, Jun Wu, Gaolei Li, Ali Kashif Bashir, Jianhua Li, and Ahmad Ali AlZubi. 2022. Recurrent semantic learning-driven fast binary vulnerability detection in healthcare cyber physical systems. *IEEE Transactions on Network Science and Engineering* (2022).

[40] Xinyang Zhang, Zheng Zhang, Shouling Ji, and Ting Wang. 2021. Trojaning language models for fun and profit. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–197.

[41] Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, and Xiangyu Zhang. 2023. {PELICAN}: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2365–2382.