Hecate: Automated Customization of Program and Communication Features to Reduce Attack Surfaces

Hongfa Xue, Yurong Chen, Guru Venkataramani, and Tian Lan

The George Washington University
{hongfaxue, gabrielchen, guruv, tlan}@gwu.edu

Abstract. Customizing program and communication features is a commonly adopted strategy to counter security threats that arise from rapid inflation of software features. In this paper, we propose **Hecate**, a novel framework that leverages dynamic execution and trace to create customized, self-contained programs, in order to minimize potential attack surface. It automatically identifies program features (i.e., independent, well-contained operations, utilities, or capabilities) relating to application binaries and their communication functions, tailors and eliminates the features to create customized program binaries in accordance with user needs, in a fully unsupervised fashion. **Hecate** makes novel use of deep learning to identify program features and their constituent functions by mapping dynamic instruction trace to functions in the binaries. It enables us to modularize program features and efficiently create customized program binaries at large scale. We implement a prototype of **Hecate** using a number of open source tools such as DynInst and TensorFlow. Evaluation using real-world executables including OpenSSL and LibreOffice demonstrates that **Hecate** can create a wide range of customized binaries for diverse feature requirements, with the highest accuracy up to 96.28% for feature/function identification and up to 67% reduction of program attack surface.

Keywords: Program customization· Deep learning · Binary analysis.

1 Introduction

Feature creep, referring to the ongoing expansion and addition of new features (e.g., excessive capabilities and utilities) in communication protocols and programs [8], leads to not only software system bloat, but also an increased attack surface with higher possibility of vulnerabilities and exploitation. A number of proposals have been made to identify redundant features and to enable customization through static code analysis techniques such as [32, 26, 15].

In this paper, we propose **Hecate**, a framework that leverages dynamic execution and trace to create customized, self-contained programs to minimize the corresponding attack surface. A key feature of **Hecate** is that it makes novel use of deep learning to identify program and communication-related features in binary in an automated fashion. It employs the test-cases to invoke different program features, applies trace splicing to extract dynamic execution paths (of invoked features) from the complete instruction trace, maps the paths to owner functions in the binary code, finally identifies program features (as targets for customization) through their constituent functions. We note that this is a challenging problem, since full symbol or debug information is often not available in optimized and obfuscated binaries, while static analysis techniques such as execution path alignment [14] cannot easily achieve scalability and accuracy. **Hecate** addresses the challenge by leveraging deep learning. In particular, we consider this mapping from execution path trace to their onwer funcsions as a multi-class classification problem, where each function is considered as a class label, the function's binary code as samples of the class, and an execution path extracted from dynamic instruction trace as the testing sample. Thus, we employ Recursive Neural Network (RNN) to obtain binary code vector embeddings at lexical level and train a multi-class Convolutional Neural Network (CNN) classifier to identify the feature-constituent functions. Instead of extracting the instructions of a limited code fragment, our approach automatically identify various features in large-scale program binaries with accuracy up to 96.28%, in a fully unsupervised fashion.

Identifying the feature-constituent functions enables us to modularize and tailor program features, in accordance with user needs. We propose program customization techniques to tailor program binaries using union, intersection, and subtraction operations if a target feature combination is not readily available in the test-cases. The customized program can be viewed as a sub-graph of the original CFG.

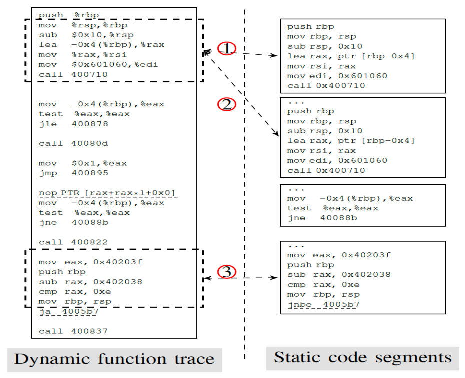
We implement a prototype of **Hecate** using two major modules: feature identification and feature tailoring. It leverages several open-source tools and deep learning algorithms to identify function boundaries and bodies from binary executable. Evaluation using real-world applications, e.g., OpenSSL [22] and LibreOffice [17], shows that **Hecate** achieves an average 92.76 % accuracy for function mapping and feature identification. It is able to create a wide range of customized executables and significantly reduces the program size and attack surface up to 85% and 67.6% respectively.

The main contributions of our work are as follows:

- We propose **Hecate**, an automated framework for software mass customization using only binaries. Provided with test-cases for different features, **Hecate** automatically identifies program features and customizes them in accordance with user needs.
- **Hecate** leverages deep learning to identify program features in an unsupervised fashion. In particular, it maps dynamic execution paths from the instruction trace to feature-constituent functions in the executable using a multi-class CNN classifier, achieving an average 92.76% accuracy.
- We implement a prototype of **Hecate** using open-source tools, including ByteWeight [3], RNNLM Toolkit [13], and Tensor-Flow [2]. Evaluation using real-world applications, such as OpenSSL, shows that **Hecate** can efficiently customize large-scale software, and significantly reduce the attack surface by up to 67%.

2 Hecate Design Overview

Software customization comprises two tasks: (i) identifying program features from a binary executable by analyzing and mapping dynamic instruction trace



```
push  %rbp
mov   %rsp,%rbp
sub   $0x10,%rsp
lea   -0x4(%rbp),%rax
mov   %rax,%rsi
mov   $0x601060,%edi
call  400710

mov   -0x4(%rbp),%eax
test  %eax,%eax
jle   400878

call  40080d

mov   $0x1,%eax
jmp   400895

nop PTR [rax+rax*1+0x0]
mov   -0x4(%rbp),%eax
test  %eax,%eax
jne   40088b

call  400822

mov eax, 0x40203f
push rbp
sub rax, 0x402038
cmp rax, 0xe
mov rbp, rsp
ja  4005b7

call  400837
```

```
push rbp
mov rbp, rsp
sub rsp, 0x10
lea rax, ptr [rbp-0x4]
mov rsi, rax
mov edi, 0x601060
call 0x400710
```

```
...
push rbp
mov rbp, rsp
sub rsp, 0x10
lea rax, ptr [rbp-0x4]
mov rsi, rax
mov edi, 0x601060
call 0x400710
```

```
...
mov   -0x4(%rbp),%eax
test  %eax,%eax
jne   40088b
```

```
...
mov eax, 0x40203f
push rbp
sub rax, 0x402038
cmp rax, 0xe
mov rbp, rsp
jnbe  4005b7
```

Fig. 1: An illustrative example of feature identification by mapping dynamic instruction trace to functions in static code from OpenSSL.

that invokes different features, and (ii) tailoring and rewriting the binary, in accordance with user needs, to create customized, self-contained programs.

2.1 Challenges

The goal of **Hecate**'s feature identification is to map dynamic instruction trace (relating to different features) to feature-constituent functions in binary. Ideally, it is possible to log the virtual addresses of each executed instruction. Then we can get the memory layout of each binary module (e.g., through /proc/pid/maps on Linux). With these two pieces of information, we could uniquely map a dynamic trace back to static code. However, there are some scenarios in practice where the address is not available. For example, commercial software and operating system are usually slightly obfuscated to deter reverse engineering and unlicensed use. Further, system and kernel libraries are often optimized to reduce disk space requirements[6]. It may be difficult to even locate function entry points (FEPs) since the full symbol or debug information is usually not available in optimized binaries [3]. Thus, we have to utilize code patterns to match dynamic traces. This is a challenging problem because dynamic trace and static code often have different patterns and cannot be accurately matched through techniques such as execution path alignment [14]. Consider the example shown in Figure 1 with dynamic instruction trace and binary code snippet from OpenSSL. First, as Arrows 1 and 2 indicate, the same basic block from dynamic instruction trace could have multiple matches in the binary, and cannot be uniquely mapped to a single function. Second, the same binary instruction can be interpreted into different verbal presentations, in which case different disassemblers will give different outputs. As Arrow 3 indicates, the binary value 77H can be translated to the opcode either "ja" (jump above) or "jnbe" (jump not below), causing direct pattern matching to fail. Further, when loops and recursive function calls exist in the binary, it is difficult to correctly identify these structures in dynamic instruction trace. We conducted an experiment using a substring matching approach to map the opcode pattern between instruction traces and binary code.

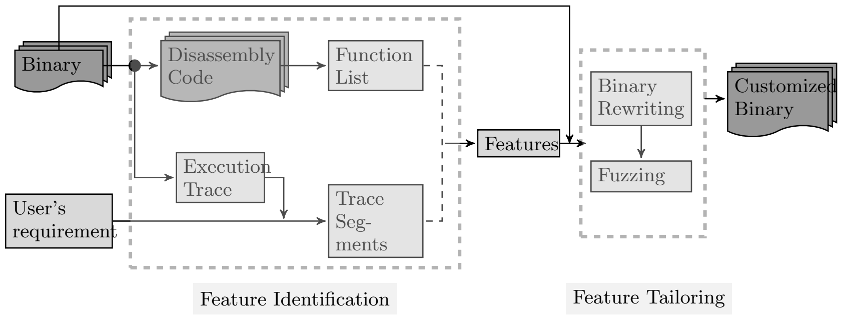


Fig. 2: **Hecate** System Diagram

Examining two applications, bzip2 and OpenSSL, function mapping techbniques only achieves an average accuracy of 76.31% and 73.02%, respectively.

2.2 Problem Statement

To introduce our problem of software customization, we first need a definition of what a feature is in binary code.

Definition 1. ***Function.*** *The term* function *in this paper particularly refers to the function identified in static binary code, which is a collection of basic blocks with one entry point (i.e., the next instruction after a call instruction) and possibly multiple exit points (i.e., a return or interrupt instruction). All code reachable from the entry point before reaching any exit point constitutes the body of the assembly function. For a given program, we use $\mathcal{F} = \{f_k,\ \forall k\}$ to denote the set of all functions existing in the static binary code.*

Definition 2. ***Feature.*** *A program feature is defined as a set of constituent functions – denoted by $F_i = \{f_i^1, f_i^2, ..., f_i^n\} \subseteq \mathcal{F}$ – which uniquely represent an independent, well-contained operation, utility, or capability of the program. A feature at the binary level may not always correspond to a software module at the source level. We use $\mathcal{T} = \{F_i,\ \forall i\}$ to denote the set of all available features in the program.*

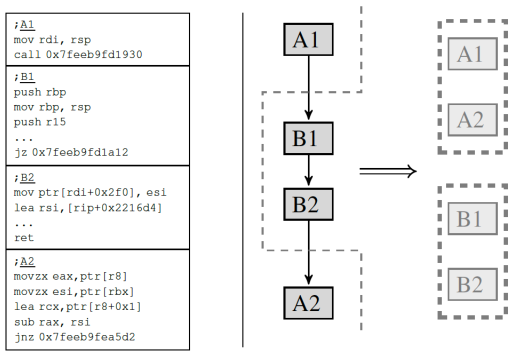
The goal of **Hecate** is that, given a program binary, test cases invoking program features, and user's customization requirement (i.e., a set of desire features $\hat{\mathcal{T}} \subseteq \mathcal{T}$), it will produce a modified binary that contains the minimum set of functions to satisfy the user's requirement and to support all desired features in $\hat{\mathcal{T}}$. We perform the customization after abstracting the program into Control Flow Graph (CFG). From the perspective of CFG, the customized binary is composed of a CFG that is a subgraph of the original program CFG.

2.3 Approach and System Architecture

Hecate consists of two major modules: feature identification and feature tailoring. Its system architecture is illustrated in Figure 2. Users provide their requirements (i.e., a list of features that are needed) as well as test-cases to reach different features. **Hecate** takes the program binary and customization requirement as inputs and generate a customized binary consisting of only the desired features. For feature identification, **Hecate** first builds a function library based on static analysis of program binary, including dynamically linked libraries. Byteweight, a learning-based binary analysis tool, is employed to identify function body directly from static program binaries. Next, execute the program using the test-cases provided, analyze the dynamic instruction trace, extract execution paths relating to different features (or feature combinations), and maps them to constituent-functions in the program binary.

The feature tailoring module is explained in section 4. It modularizes program features through their constituent functions and modifies the program binary in accordance with user's customization requirements. The CFG of the customized program can be viewed as a sub-graph of that of the original program, which is able to retain the behavior of only the desired features. At last, a fuzzing engine can be employed to generate inputs and further test the customized binary.

3 Feature Identification



```
;A1
mov rdi, rsp
call 0x7feeb9fd1930
;B1
push rbp
mov rbp, rsp
push r15
...
jz 0x7feeb9fd1a12
;B2
mov ptr[rdi+0x2f0], esi
lea rsi,[rip+0x2216d4]
...
ret
;A2
movzx eax,ptr[r8]
movzx esi,ptr[rbx]
lea rcx,ptr[r8+0x1]
sub rax, rsi
jnz 0x7feeb9fea5d2
```

Fig. 3: Extracting dynamic execution paths of each individual function through trace splicing. Boxes stand for basic blocks. A_1 and A_2 belong to function A while B_1 and B_2 belong to function B.

Feature Identification uses trace splicing to extract dynamic execution paths and maps them to owner functions in the binary code, enabling us to identify program features through their constituent functions. In this paper, we define an *execution path* as a sequence of instructions that are executed from a function entry point to an exit point. The function containing the execution path is known as the *owner function*. Our approach leverages deep learning and works in a fully unsupervised, autonomous fashion.

3.1 Function Recognition

We first construct the pre-image and image of our function mapping, using trace splicing and deep-learning tools, respectively. The pre-image is defined as the set of execution paths obtained from dynamic instruction trace, while the image is defined as the set of functions recognized in static program binaries.

We run the target executable with provided test cases to invoke different (combinations of) program features, and collect instruction trace to capture the dynamic execution of the program. The trace is then spliced to extract execution paths belonging to different functions, which serves as the pre-image of our function mapping. Consider the illustrative example shown in Figure 3, where a sequence of 4 basic blocks, A_1, B_1, B_2, A_2, are captured in dynamic trace, when function f_B is called inside function f_A. Clearly, we cannot directly map the entire sequence to functions in binary code, because it contains two separate execution path, belonging to functions f_A and f_B, respectively. We employ two different methods to splice dynamic trace and extract different execution paths: (1) We track call stack changes together with instruction trace. By recognizing *push* and *pop* operations on the call stack, we can infer function call events, and slice and associate basic blocks that belong to the same function. (2) From the instruction trace, instructions that perform function calls and returns will be recognized and put embedded function calls into different layers.

We remove duplicate basic blocks in execution traces to improve the accuracy of function mapping. Furthermore, every time a function is invoked, a different execution path may be traversed inside the function. These execution paths will be separated and mapped to their owner functions independently, minimizing the probability of false negative in function mapping. In this paper, we unitize ByteWeight [3], a learning-based tool that identifies function bodies from binary.

3.2 Function Mapping

In this paper, we leverage deep learning to propose a solution to enable automated function mapping. model binary instruction sequences using Recursive Neural Network (RNN). The framework is constructed with two key components. First, to obtain vector embedding for a given execution path (that consists of an instruction sequence), we use RNN to map each term in the binary instructions (e.g., opcodes and operands) to a vector embedding at the lexical level, resulting in a signature vector for the entire execution path. Second, we consider the mapping problem as a multi-class classification problem, where each function is considered as a class label, different execution paths obtained from the function's binary code as samples of that class, and an execution path extracted from dynamic instruction trace as the testing sample. We employ a multi-class Convolutional Neural Network (CNN) classifier to identify the owner functions of an arbitrary dynamic instruction trace. Our deep learning approach is inspired by the related work on source code analysis [24, 30, 25]

Embedding binary code at the lexical level. Consider a disassembly code corpus from a target program, with m distinct terms (e.g., different opcodes and operands) across the whole corpus. We use an RNN with n hidden nodes to convert each term in the code corpus into an embedding vector $U \in \mathbb{R}^{n\times m}$. RNN

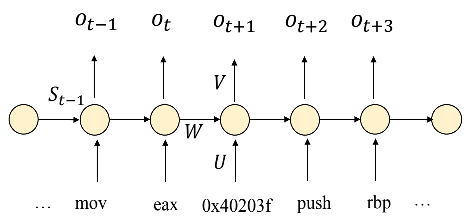
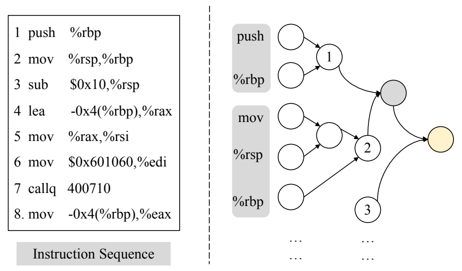


Fig. 4: An illustration of RNN.



```
1 push   %rbp
2 mov    %rsp,%rbp
3 sub    $0x10,%rsp
4 lea    -0x4(%rbp),%rax
5 mov    %rax,%rsi
6 mov    $0x601060,%edi
7 callq  400710
8. mov   -0x4(%rbp),%eax
```

Fig. 5: An Illustration of RAE.

is known as an effective approach for modeling sequential information, such as sentences in texts or program code. Figure 4 presents the training process of our RNN model for binary code. The input $x_t \in \mathbb{R}^{m+n}$ at time step t is a one-hot vector representation corresponding to the current term, e.g., *'eax'*. The hidden layer state vector, $s_t \in \mathbb{R}^n$, stores the current state of the network at step t and captures the information that has already been calculated. Specifically, it can be obtained using the previous hidden state s_{t-1} at time step $t-1$ and the current input x_t at time step t:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{1}$$

Function f is a nonlinear function, e.g., *tanh* [12]. $U \in \mathbb{R}^{n\times m}$ and $W \in \mathbb{R}^{n\times n}$ are the shared parameters in all time steps.

The output, $O_t \in \mathbb{R}^m$, is a vector of probabilities predicting the distribution of the next term in the code corpus. It is calculated based on current state vector along with another shared parameter $V \in \mathbb{R}^{m\times n}$, i.e., :

$$O_t = softmax(Vs_t) \tag{2}$$

The parameters $\{U, V, W\}$ are trained using backpropagation through time (BPTT) method in our RNN network (We skip the technical details here and refer readers to [4]). Once RNN training is complete, each term in the code corpus will have a unique embeddings U from Equation (1), which comprises its semantic representation cross the corpus. We compute such embeddings U to represent the terms of binary instructions at lexical level.

Generating signature at the syntax level. We use Autoencoder to combine embeddings $U \in \mathbb{R}^{nm}$ of the terms from multiple instructions and to obtain a signature vector for a given execution path. Autoencoder is widely used to generate vector space representations for a pairwise composed term with two phases: encode phase and decode phase. It is a simple neural network with one input layer, one hidden layer, and one output layer. As shown in Figure 5, we apply Autoencoder recursively to a sequence of terms, which is known as the Recursive Autoencoder (RAE). Let $x_1, x_2 \in \mathbb{R}^{nm}$ be the vector embeddings of two different terms, computed using RNN. During encode phase, the composed vector embeddings $Z(x_1, x_2)$ is calculated by:

$$Z(x_1, x_2) = f(W_1[x_1; x_2] + b_1), \tag{3}$$

where $[x_1; x_2] \in \mathbb{R}^{2nm}$ is the concatenation of x_1 and x_2, $W_1 \in \mathbb{R}^{nm \times 2nm}$ is the parameter matrix in encode phase, and $b \in \mathbb{R}^{nm}$ is the offset. Similar to RNN, f again is a nonlinear function, e.g., *tanh*. In decode phase, we need to assess if $Z(x_1, x_2)$ is well learned by the network to represent the composed terms. Thus, we reconstruct the the term embeddings by:

$$O[x_1; x_2] = g(W_2[x_1; x_2] + b_2), \tag{4}$$

where $O[x_1; x_2]$ is the reconstructed term embeddings , $W_2 \in \mathbb{R}^{nm \times 2nm}$ is the parameter matrix for decode phase, and $b_2 \in \mathbb{R}^{nm \times 1}$ is the offset for decode phase and the function g is another nonlinear function. For training purpose, the reconstruction error is used to measure how well we learned term vector embeddings. Let $\theta = \{W_1; W_2; b_1; b_2\}$. We use the Euclidean distance between the inputs and reconstructed inputs to measure reconstruction error, i.e.,

$$E([x_1; x_2]; \theta) = ||[x_1; x_2] - O[x_1; x_2]||_2^2 \tag{5}$$

For a given execution path with multiple terms and instructions, we adopt a greedy method [23] to train our RAE and recursively combine pairwise vector embeddings. The greedy method uses a hierarchical approach – it first combines vector embeddings of adjacent terms in each instruction, and then combines the results from a sequence of instructions in an execution path. Figure 5 shows an example of how to combine the vector embeddings to generate a signature vector. It shows a (binary) execution path with a sequence of 8 instructions. The greedy method is illustrated as a binary tree. Node 1 gives the vector embedding for the first instruction $Inst_1 = (push\ \%rbp)$ encoded from terms $[push; \%rbp]$. Then, we continue to process the remaining instructions, e.g., Nodes 2 and 3, until we derive the final vector embedding (i.e., the signature vector) for the instruction sequences of the given execution path.

Multi-class classification for function mapping. Function mapping aims to recognize the owner function (in static binary) of a given execution path obtained from the dynamic trace. We consider each function as a class label, different execution paths obtained from the function binary code as samples of that class, and an execution path extracted from dynamic instruction trace as the testing sample. Then, the mapping becomes a multi-class classification problem, which is solved using Convolutional Neural Networks (CNN) in this paper. We adopt the sentence classification model proposed in [9, 33] for natural language processing and train a multi-class classifier using CNN for function mapping. Note that another line of work, such as tainting [31, 19], can be used for feature identification. We consider this as future work.

To obtain training samples for each class, we use CFG analysis to construct different execution paths for each function identified in the binary code. More precisely, once the function boundaries and bodies are recognized, we use a Depth First Search (DFS) to traverse the static CFG of each function and construct related execution path using a *random walk*.

4 Feature Tailoring

Feature tailoring creates customized software that consists of the desired features and their constituent functions in accordance with user needs. It has to address a number of challenges. First, a single execution trace may not reach all desired features, requiring us to merge multiple outputs from feature identification. Second, different features often share some common constituent functions. If the goal of tailoring is to remove certain features, we need to identify and retain the shared functions in the customized binary.

4.1 Feature Tailoring

Let $\hat{\mathcal{F}}$ be a set of target program features for tailoring. If the constituent functions of each feature $F_i \in \hat{\mathcal{F}}$ can be successfully identified, we can simply create a superset of their constituent functions, i.e., $\hat{F} = \cup F_i$. Two techniques are developed next to (i) create a customized program by retaining only the features in $\hat{F}$ (e.g., if user only needs these features) and (ii) remove the features in $\hat{F}$ from the binary (e.g., if they are deemed as unnecessary or vulnerable). When $\hat{F}$ cannot be directly identified, we leverage set operations, including union, intersection, and subtraction, to construct $\hat{F}$ from available feature combinations, in order to fulfill feature tailoring.

Tailoring via set operations. When the target features' constituent functions $\hat{F}$ are not directly identifiable, **Hecate** employs set operations including union, intersection, and subtraction to compute $\hat{F}$ from known feature combinations. **Union:** A feature may contain multiple execution paths that cannot be dumped and identified in a single execution. **Hecate** will collect traces from different program executions to identify and compute the union of the related feature-constituent functions. **Intersection:** A program may contain concurrent features that cannot be identified separately from the available execution trace. For instance, OpenSSL's *choosing cipher suite* feature is always coupled with the execution of encryption/hash functions in dynamic trace. To identify the constituent functions of *choosing cipher suite* feature, we can take the intersection of multiple executions with different choices of encryption/hash functions. **Subtraction:** This operation allows us to identify the unique constituent functions of given features. So, we can safely remove them without affecting the soundness of other features due to shared functions.

4.2 Binary Rewriting

We use feature tailoring to derive a set of functions to eliminate in program binary. Simply replacing these function bodies with "NOP"s would not generate a valid executable, because (i) some code segments in the eliminated function body may be shared with other functions, and (ii) there may exist data segments that are inserted into the eliminated functions and must be preserved.

To address these issues, **Hecate** utilizes a static binary rewriter, DynInst, to modify the program binary by rewriting the binaries in basic blocks level in the CFG. As DynInst is capable to abstract the program basic blocks in the form of CFG. To remove the features in the programs, there are two steps in **Hecate**˙First, **Hecate** removes the functions that should not be called. The call site of the eliminated functions will be replaced to redirect the program to exit point. Second, for those functions cannot be removed from the first step

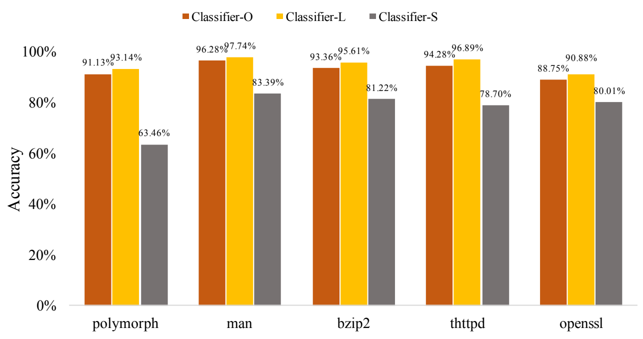


Fig. 6: Accuracy of function mapping during feature identification

(e.g., For indirect function calls, the address of the callee function cannot be decided beforehand and can potentially lead to any other addresses), we replace the rest of the function body with "NOP". Furthermore, a verification process is performed using program fuzzing approaches [32] by **Hecate** to validate the effectiveness and correctness of feature tailoring.

5 Evaluation

5.1 Experiment Setup

Our experiments are conducted on a 2.80 GHz Intel Xeon(R) CPU E5-2680 20-core server with 16 GByte of main memory. The operating system is Ubuntu 14.04 LTS.

Benchmarks. In our evaluation, we select three sets of real-world applications: (i) Non-interactive applications including two applications from SPEC 2006 Benchmark suite [1], bzip2 and hmmer; two applications from a bug benchmark suite *bugbench* [11], polymorph and man and (ii) Interactive applications including a light-weight web server thttpd, version beta 2.23, an open source office suite LibreOffice and a web browser links. (iii) An implementation of Transport Layer Security (TLS) & Secure Sockets Layer (SSL) protocol, OpenSSL.

Dataset and Training. In our function mapping module, we collect static execution paths as training dataset and dynamic execution paths as testing dataset for evaluating the accuracy of the pre-trained models. We selected the highest quality model and extracted the matrix of embeddings. We have observed that a well trained function mapping model is with the hidden node size as 500 in RNN and 200 maximum iterations for RAE, which is chosen as the parameters of deep neural network in function mapping module.

5.2 Accuracy of function mapping

In this section, we evaluate the accuracy of the pre-trained function mapping module in **Hecate** and presents the accuracy of five representative applications. We construct the testing dataset as follows: We collect the dynamic instruction traces for each identified function in the binary and perform the same *random walk* process to generate execution paths as mentioned in Section 3.2. The testing dataset size is controlled to be 30% as big as the training dataset We also observed that due to the different amount of training data we can obtain from different functions, the mapping accuracy will be higher if we split functions into large and small categories, by using the median number of training data sample size. We trained three CNN classifiers for each application, one is trained cross all the functions as an overall classifier (Classifier-O), and the other two are trained for large functions (Classifier-L) and small functions (Classifier-S) respectively.

The function mapping accuracy is plotted in Figure 6. We achieve an overall average accuracy of 92.76%, with the highest up to 96.28% in *man* from *bugbench*. In general, the mapping accuracy of larger programs, such as bzip2 and thttpd, is higher than smaller programs like polymorph. Because the number of execution traces used for training our CNN classifiers in those programs is much larger than that in polymorph, there are 189,855 training execution paths in bzip2 comparing to 10,806 in polymorph). For the applications with more functions, such as OpenSSL that has 4,023 functions, the overall accuracy can be as low as 88.75% since there are more classes for classification. We also note that all of the Classifier-Ls outperforms the Classifier-Os. For instance, in *polymorph*, the accuracy of Classifier-L is 93.14% whereas the accuracy of Classifier-O is 91.13%. However, we observe that the accuracy for Classifier-S is lower than Classifier-L. The reason is that functions trained in Classifier-Ss are relatively small, with limited training data samples for classification. In particular, the accuracy of Classifier-S is 63.46% for polymorph, which is the worst among all the applications. We further analyzed and found that the median number of training data size is 7 for polymorph, which means almost half of the functions have only less than 7 training data samples. The lack of training data leads to a bad performance for classification.

5.3 Impact on program security

We evaluate the impact of feature customization on program security here. As shown previously, the reduction of code size also shrink the attack surface and eliminate possible vulnerabilities in programs. We survey the known CVEs of different programs that can be removed by feature customization. For instance, in OpenSSL, i) the *CVE-2014-0160*, known as *Heartbleed* bug, can be eliminated by removing the *heartbeat* extension; ii) the *CVE-2016-7054*, which can lead to DoS attack can be neutralized by removing **-CHACHA20-POLY1305* ciphersuites; iii) the *CVE-2016-0701*, which can cause information leakage, can be negated by avoiding using DH ciphersuites; The *CVE-2015-5212* in LibreOffice (an integer underflow bug) can be removed by disabling the printer functionality when users don't need it.

In total. we found 101 CVEs in OpenSSL distributions during 2014-2017, 34 CVEs in LibreOffice, 13 CVEs in Thttpd and 9 CVEs in Bzip2. Not all vulnerabilities can be disabled by our feature customization. Some vulnerabilities are in the functions that are necessary for program execution. *CVE-2010-0405* in Bzip2 is an integer overflow bug in function *BZ2_decompress*. In most of the

Program	# Removed CVEs	% Features removed
OpenSSL(2014-2017)	45	44.6
LibreOffice	23	67.6
Thttpd	5	38.5
Bzip2	2	22.2

Table 1: Impact on Application and Communication security

cases, decompression is a feature that users will not remove. The number and ratio of program features that can be removed are shown in Table 1. We evaluate the security impact of **Hecate** using the ratio of CVEs that can be removed by feature customization.

6 Related Work

Code analysis and De-bloating: Several prior works have proposed program customization frameworks only based one methods like de-bloating [7], cross-host tainting [5] and so on. In terms of binary reuse, it has been studied by several works [27, 28]. The main challenge of reusing binary code is it only focuses on reusing partial code in the program high-level assembly code. Some existing works try to find memory-related vulnerabilities in source code or IR by direct static analysis [29, 20, 21]. As such, the two approaches are quite complementary and when combined together, can present an improved framework for eliminating attack surfaces in programs.

Learning-based approach for vulnerability removal: Prior work has studied bug/vulnerabilities removal using learning-based approaches. StatSym [30] and SARRE [10] propose frameworks combining statistical and formal analysis for vulnerable path discovery. SIMBER [25] proposes a statistical inference framework to eliminate redundant bound checks and improve the performance of applications without sacrificing security.

7 Conclusion, Future work and Opportunities

In this paper, we design and evaluate a binary customization framework **Hecate**, that aims to generate customized program binaries with *just-enough* features and can satisfy a broad array of customization demands. Feature identification and feature tailoring are two major modules in **Hecate**, with the former one discovering the target features using both static code and execution traces, and the latter one modifying the features to reconstruct a customized program. Our experiment results demonstrate that **Hecate** is able to identify features with the highest accuracy up to 96.28% and reduce the attack surface by up to 67%.

Generating test cases to cover all corner cases of a feature is a challenging problem in general. To deal with this problem, we note that some approaches, such as fuzzing techniques [18], can be useful. As reported in Section 5, our deep learning-based function mapping model achieves an average accuracy of

92.7%. However, we could increase the training data size by collecting the dynamic execution paths and use related machine learning optimization like cross-validation to split small data set [16] for further performance improvements. Moreover, more complex deep learning algorithms can be further tested, such as bi-directional RNN and long-short-term memory (LSTM), which have been proven a better performance for modeling longer sequential information. We will consider the above concerns as our future work.

Acknowledgments

This work was supported by the US Office of Naval Research (ONR) under Awards N00014-15-1-2210 and N00014-17-1-2786. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

References

1. Spec cpu 2006. `https://www.spec.org/cpu2006/`
2. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI (2016)
3. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: Byteweight: Learning to recognize functions in binary code. USENIX (2014)
4. Bishop, C.M.: Machine learning and pattern recognition. Information Science and Statistics. Springer, Heidelberg (2006)
5. Chen, Y., Sun, S., Lan, T., Venkataramani, G.: Toss: Tailoring online server systems through binary feature customization. In: FEAST workshop (2018)
6. Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. ACM SIGARCH Computer Architecture News (2005)
7. Jiang, Y., Wu, D., Liu, P.: Jred: Program customization and bloatware mitigation based on static analysis. In: IEEE Computer Software and Applications Conference (2016)
8. Jiang, Y., Zhang, C., Wu, D., Liu, P.: Feature-based software customization: Preliminary analysis, formalization, and methods. In: High Assurance Systems Engineering (2016)
9. Kim, Y.: Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882 (2014)
10. Li, Y., Yao, F., Lan, T., Venkataramani, G.: Sarre: semantics-aware rule recommendation and enforcement for event paths on android. IEEE Transactions on Information Forensics and Security (2016)
11. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: Bugbench: Benchmarks for evaluating bug detection tools. In: Workshop on the evaluation of software defect detection tools (2005)
12. Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Annual Conference of the International Speech Communication Association (2010)
13. Mikolov, T., Kombrink, S., Deoras, A., Burget, L., Cernocky, J.: Rnnlm-recurrent neural network language modeling toolkit. In: ASRU Workshop (2011)
14. Ming, J., Xu, D., Jiang, Y., Wu, D.: Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In: USENIX Security (2017)

15. Oh, J., Hughes, C.J., Venkataramani, G., Prvulovic, M.: Lime: A framework for debugging load imbalance in multi-threaded execution. In: Proceedings of the 33rd International Conference on Software Engineering. ACM (2011)
16. Smith, G.C., Seaman, S.R., Wood, A.M., Royston, P., White, I.R.: Correcting for optimistic prediction in small data sets. American journal of epidemiology (2014)
17. Source, O.: Libreoffice
18. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS (2016)
19. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Flexitaint: A programmable accelerator for dynamic taint propagation. In: IEEE International Symposium on High Performance Computer Architecture (2008)
20. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Memtracker: An accelerator for memory debugging and monitoring. ACM Transactions on Architecture and Code Optimization (TACO) (2009)
21. Venkataramani, G., Hughes, C.J., Kumar, S., Prvulovic, M.: Deft: Design space exploration for on-the-fly detection of coherence misses. ACM Transactions on Architecture and Code Optimization (TACO) (2011)
22. Viega, J., Messier, M., Chandra, P.: Network Security with OpenSSL: Cryptography for Secure Communications. " O'Reilly Media, Inc." (2002)
23. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: IEEE/ACM Intl Conference on Automated Software Engineering (2016)
24. Xue, H., Chen, Y., Venkataramani, G., Lan, T., Jin, G., Li, J.: Morph: Enhancing system security through interactive customization of application and communication protocol features. In: Poster in ACM Conference on Computer and Communications Security (2018)
25. Xue, H., Chen, Y., Yao, F., Li, Y., Lan, T., Venkataramani, G.: Simber: Eliminating redundant memory bound checks via statistical inference. In: IFIP SEC (2017)
26. Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: A comprehensive study. IEEE Access (2019)
27. Xue, H., Venkataramani, G., Lan, T.: Clone-hunter: accelerated bound checks elimination via binary code clone detection. In: ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (2018)
28. Xue, H., Venkataramani, G., Lan, T.: Clone-slicer: Detecting domain specific binary code clones through program slicing. In: FEAST workshop. ACM (2018)
29. Yao, F., Chen, J., Venkataramani, G.: Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In: 2013 IEEE 31st International Conference on Computer Design (ICCD). IEEE
30. Yao, F., Li, Y., Chen, Y., Xue, H., Lan, T., Venkataramani, G.: Statsym: vulnerable path discovery through statistics-guided symbolic execution. In: Dependable Systems and Networks (DSN) (2017)
31. Yao, F., Venkataramani, G., Doroslovački, M.: Covert timing channels exploiting non-uniform memory access based architectures. In: Great Lakes Symposium on VLSI. ACM (2017)
32. Zalewski, M.: American fuzzy lop (2007)
33. Zhang, K., Wang, M., Cong, X., Huang, F., Xue, H., Li, L., Gao, Z.: Personal attributes extraction based on the combination of trigger words, dictionary and rules. In: Proceedings of The Third CIPS-SIGHAN Joint Conference on Chinese Language Processing. pp. 114–119 (2014)