EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

AMASS: Automated Software Mass Customization via Feature Identification and Tailoring

Hongfa Xue[1,*], Yurong Chen[1], Guru Venkataramani[1], Tian Lan[1]

[1]The George Washington University, 800 22nd Street NW, Washington, DC, 20056, USA

Abstract

The rapid inflation of software features brings inefficiency and vulnerabilities into programs, resulting in an increased attack surface with a higher possibility of exploitation. In this paper, we propose a novel framework for automated software mass customization (**AMASS**), which automatically identifies program features from binaries, tailors and eliminates the features to create customized program binaries in accordance with user needs, in a fully unsupervised fashion. It enables us to modularize program features and efficiently create customized program binaries at large scale. Evaluation using real-world executables including OpenSSL and LibreOffice demonstrates that **AMASS** can create a wide range of customized binaries for diverse feature requirements, with an average 92.76% accuracy for feature/function identification and up to 67% reduction of program attack surface.

Received on 30 January 2019; accepted on 08 April 2019; published on 29 April 2019

Keywords: Program customization, Deep learning, Binary analysis.

doi:10.4108//eai.13-7-2018.162291

1. Introduction

Feature creep [11], which refers to the ongoing expansion and addition of new features (e.g., excessive capabilities and utilities), is becoming increasingly commonplace for software development in most commercial applications. It results in not only larger installation footprint, but also an increased attack surface with higher possibility of vulnerabilities and exploitation, as evidenced by recent security breaches like attacks related to OpenSSL's keep-alive feature (i.e., Heartbleed [7]) and Struts' file upload feature (i.e., Jakarta Multipart Parser OGNL Injection [21]). A commonly adopted solution to counter such security attacks is to minimize the attack surface by creating customized software systems that contain *just-enough* features and yet satisfy specific customer needs. However, it is important to note that delivering such customized software for diverse use-cases is currently an extremely slow, built-to-order process, and on occasions, an impossible task.

Main problem and challenges. In this paper, we propose a new approach for *automated software mass customization* (AMASS), which can be defined as the process of automatically identifying and tailoring different software features from a binary executable, so that customized programs can be created in high volume and at relatively low cost to better match the diverse customer needs while minimizing unwanted exploitation of the applications features. In many commercial off-the-shelf and legacy software, source code may no longer be available. Hence customization of binary is more relevant. Without requiring any knowledge of potential exploits, the customized programs will contain *just-enough* software features to support specific use-cases, thus significantly reducing the attack surface and the exposure to future exploitation through features (e.g., zero-day attacks). Our approach goes beyond existing work on feature separation [22], reduction [11] and code de-bloating [9, 36, 37], which focus on removing unused or unnecessary code. We argue that vigilantly managing and customizing permitted features is crucial for delivering improved software security [11]. For example, while system logging and banking utilities are both necessary, permitted features in an online banking app, studies show that 40% of iOS banking apps leak sensitive data through system logs [23]. Further, the usefulness of program features is often difficult to determine a priori. It is shown that 83% of available browser features are executed on less than 1% of the most popular 10,000 websites [26]. These features constitute a significant source of bloat and can be eliminated through browser customization to match individual customer needs.

*Corresponding author. Email: hongfaxue@gwu.edu

At the core of automated software customization, a key problem is to identify and modularize software features directly from the program's binary, without access to source code or debug symbol information. In structured programmings, a function is often the smallest unit of implementing a program feature. We define a *feature* as a collection of program functions, which uniquely represent an independent, well-contained operation, utility, or capability of the program. Identifying features from the program binary enables the modularization of various program utilities, to pave the path for feature tailoring and customization. This can be an extremely challenging problem because program features often traverse multiple functions across different regions of the binary, with certain functions that are referenced using function pointers that cannot be resolved statically. Furthermore, binary code is not designed to be modular even if its corresponding source code adopts a modular structure. A feature implemented in binary may not correspond to a unique code fragment/module that is separated from others, due to shared code.

AMASS solution. We consider that only program binary is available for customization. If some specific input is needed to reach a feature, we assume we are provided with such test-cases to execute the program. **AMASS** leverages deep learning to automatically identify program features in binary. We use the test-cases to invoke different program features, apply trace splicing to extract dynamic execution paths (of invoked features) from the instruction trace, and map them to owner functions in the binary code, in order to identify program features through their constituent functions. In particular, we consider this mapping problem 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 (e.g., in binary code reuse [5, 49]), the approach in this paper enables us to automatically identify various program features in large-scale program binaries with an average of 92.7% accuracy, 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 (i) retain a set of desired features, (ii) remove a set of unwanted features, and (iii) tailor program binaries using union, intersection, and subtraction operations if a target feature combination is not readily available in the test-cases. A static binary rewriter, DynInst, is employed to create the customized binary by eliminating unwanted feature-constituent functions and redirecting the call sites toward appropriate exit points. The customized program can be viewed as a sub-graph of the original CFG. Finally, to ensure its soundness, we perform reachability analysis on the customized program's CFG to validate the feasibility of customized binary and to rectify possible inaccuracy during deep-learning-based feature identification.

Implementation and Evaluation. We design and implement a prototype of **AMASS** with two major modules: feature identification and feature tailoring. It leverages several open-source tools and deep learning algorithms. In particular, ByteWeight [3] is used to identify function boundaries and bodies from binary executable. We develop a python script to tokenize the disassembly code and use the RNNLM Toolkit [19] and TensorFlow [2] to implement and train RNN and CNN multi-class classifier, respectively. Finally, we instrumented a binary analysis framework angr [24] for binary CFG and reachability analysis. Evaluation using real-world applications, e.g., OpenSSL [33] and LibreOffice [27], shows that **AMASS** 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:

(i) We propose **AMASS**, an automated framework for software mass customization using only binaries. Provided with test-cases for different features, **AMASS** automatically identifies program features and customizes them in accordance with user needs.

(ii) **AMASS** leverages a combined method through deep learning and CFG analysis 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.

(iii) We implement a prototype of **AMASS** using open-source tools, including ByteWeight [3], RNNLM Toolkit [19], and Tensor-Flow [2]. Evaluation using real-world applications, such as OpenSSL, shows that **AMASS** can efficiently customize large-scale software, and significantly reduce the attack surface by up to 67%.

2. AMASS Design Overview

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

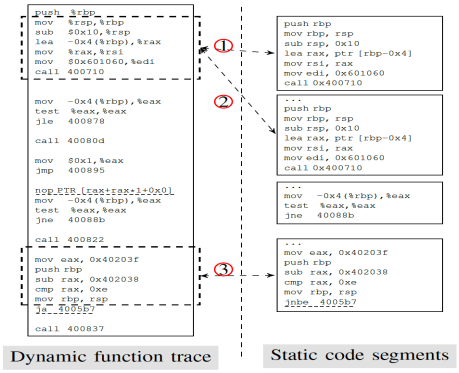

Figure 1. An illustrative example of feature identification by mapping dynamic instruction trace to functions in static code from OpenSSL.

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 **AMASS**'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/ma ps 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[8]. 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 [20]. 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. 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 and Scope

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

Definition 2.1. 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 2.2. 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.

Problem Statement: The goal of **AMASS** 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.

We provide an illustrative example of software feature customization, using a partial snapshot of LibreOffice application [27]. Figure 2 shows the subgraph of LibreOffice's CFG, which contains 3 distinct features, F_1-*save as file*, F_2-*email*, and F_3-*print*. The goal of **AMASS**'s feature identification is to find the constituent functions of each feature, as shown in Figures 3-5, respectively. Since now all three features can now be modularized through their constituent functions, we can construct a customized program containing any subset of desired features. For instance,

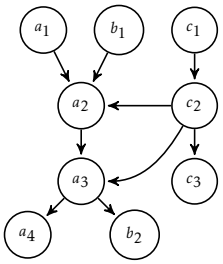

Figure 2. LibreOffice's partial CFG

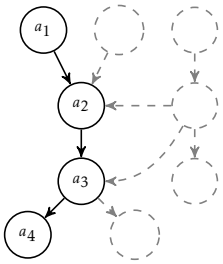

Figure 3. Feature F_1: *save as file*

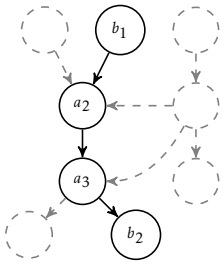

Figure 4. Feature F_2: *email*

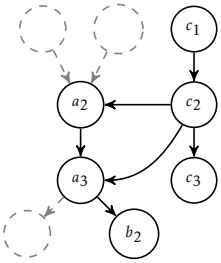

Figure 5. Feature F_3: *print*

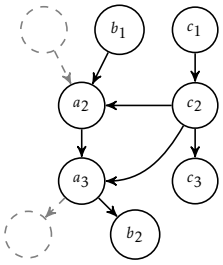

Figure 6. Feature $F_2 \cup F_3$: *email+print*

to eliminate feature F_1: *save as file*, we can tailor a program that only contains functions in$F_2 \cup F_3$, which is illustrated in Figure 6.

Scope. In this paper, we assume that only program binary is available for customization. We use a previously proposed technique, ByteWeight [3], to automatically identify function boundaries and bodies in binary with high accuracy. Features required in a target use-case is known. If some specific input is needed to reach a feature, we assume we are provided with such test-cases to execute the program. Finally, if two software utilities must always be invoked simultaneously, we consider them as a single program feature for the purpose of customization.

2.3. Approach and System Architecture

AMASS consists of two major modules: feature identification and feature tailoring. Its system architecture is illustrated in Figure 7. Users provide their requirements (i.e., a list of features that are needed) as well as test-cases to reach different features. **AMASS** takes the program binary and customization requirement as inputs and generates a customized binary consisting of only the desired features. For feature identification, **AMASS** 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. In particular, the mapping problem is solved via a multi-class classification. We leverage Recursive Neural Network (RNN) and train a multi-class Convolutional Neural Network (CNN) classifier to identify the feature-constituent functions. Thus, the output of feature identification is a set $\{F_i, \forall i\}$ of features that are identified, together with the constituent functions of each feature F_i. The details are discussed in section 3.

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. To ensure the soundness of our customization, We perform reachability analysis as described in 4.2 to validate the feasibility of the customized program by analyzing its sub-graph CFG. At last, a fuzzing engine can be employed to generate inputs and further test the customized binary.

3. Feature Identification

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*

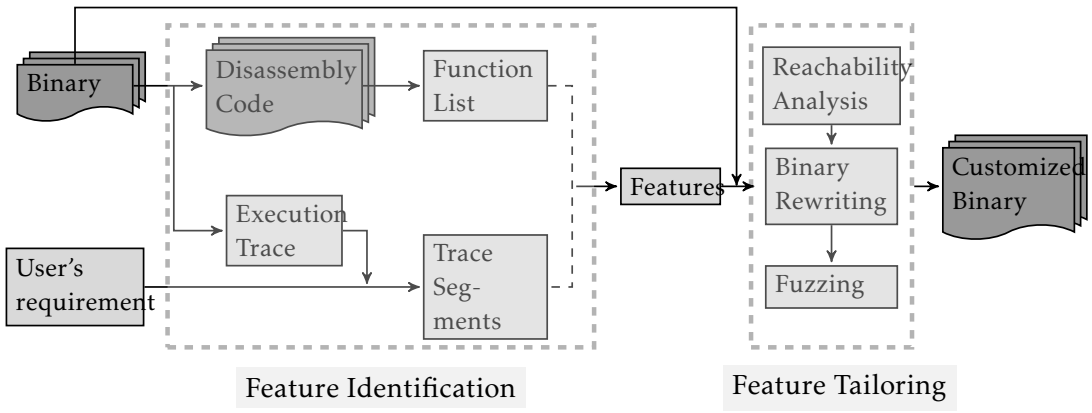

Figure 7. AMASS System Diagram

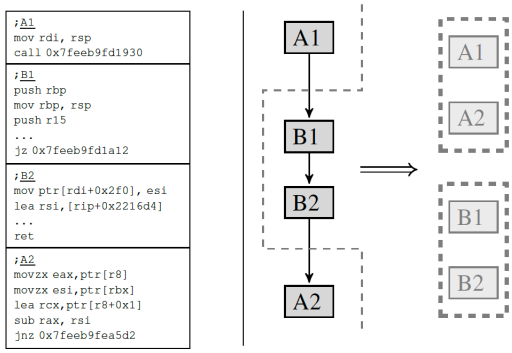

Figure 8. Extracting dynamic execution paths of each individual function through trace splicing. Boxes stand for basic blocks. A_1 and A_2 belong to function A while B_1 and B_2 belong to function B.

function. In contrast with prior work that requires seed information to bootstrap the feature identification [10]. 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.

Trace splicing: 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 8, 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. Basic blocks will be reorganized by the layers they reside along with the control flow. In particular, function entries include *call* instruction and *longjmp* while function exits include *ret*, *syscall* and interruptions. Figure 8 illustrates that after layers are appropriately identified, we extract two execution paths, A_1, A_2 and B_1, B_2, from the dynamic trace. Then, each execution path will be separately mapped to its owner function in binary.

Due to loops or repeated function calls, execution traces usually contain a large amount of duplicate basic blocks, which do not provide additional useful information for function mapping. 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. **Recognizing Static functions:** Functions in program binaries are not as easy to identify as in high-level languages, where the function boundary and prototype are well defined. Moreover, the various optimization techniques employed by compilers can make binary code difficult to construe [3]. Functions can also be separated by data or code segments from

other functions, which makes the function bodies non-contiguous. In this paper, we unitize ByteWeight [3], a learning-based tool that identifies function bodies from binary. At a high level, ByteWeight learns the signatures of function start in a weighted prefix tree during the training stage. Instructions are abstracted as nodes in the prefix tree and the edges represent the flow of instruction sequences. Each edge uses weight to indicate the confidence of current instruction sequence (from the tree root to the end node of current edge) being a function start. Afterwards, the testing code segments are matched to the signatures to get the possibility of being a starting point. After function start identification, ByteWeight utilizes CFG and value set analysis to extract function bodies corresponding to the function start points. For each target program, we build its own training data and perform the function recognition separately using ByteWeight. This allows us to better tune the learning parameters for programs that may have different structures/styles of binary code due to compilation. After training, ByteWeight operates on the target program and generates the function body and boundary information that provides a complete list of functions, as the image our function mapping.

3.2. Function Mapping

Mapping the execution paths in dynamic instruction trace to the image functions in static binary is a challenging problem. A basic block from dynamic trace could match multiple locations in binary, as they may share the same opcode sequence/pattern. Since commercial software and operating system are usually slightly obfuscated and optimized to deter reverse engineering, address information and symbol table are not always available so that we can not map a dynamic trace back to static code based one memory layout information. It also requires to cope with different verbal presentations of instructions and the existence of diverse execution paths of each function in dynamic trace. In the paper, we assume that we are provided with user-guided test-cases to execute the program and invoke target features for customization. Then we can get the Input of features execution trace after the test-cases are executed. Such test-cases can also be obtained using tools like NeuFuzz [34] and Intel PT technology with constructed Proof of Concepts (POCs) [13].

The most common approach is using Pin tool [17], a dynamic instrumentation to extract the execution path. However, it may incur a high overhead. Here, we adopt the method proposed in NeuFuzz [34], which uses Intel PT technology with constructed Proof of Concepts (POCs), which has shown a better performance in terms of execution speed.

In this paper, we leverage deep learning to propose a solution to enable automated function mapping. Deep neural network, such as Recursive Neural Network (RNN) is known as an effective approach for modeling sequential information, such as sentences in texts or programming language in source code. Similarly, one single binary instruction code s is a combination of instruction type and the corresponding operands, such as memory references, registers and immediate values, which can be considered as a sequence of tokens. We propose an approach to 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 [39, 40, 46]

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 is known as an effective approach for modeling sequential information, such as sentences in texts or program code. Figure 9 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., $tanh$ [18]. $U \in \mathbb{R}^{n \times m}$ 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.,:

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

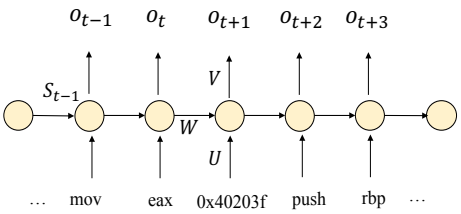

Figure 9. An illustration of RNN.

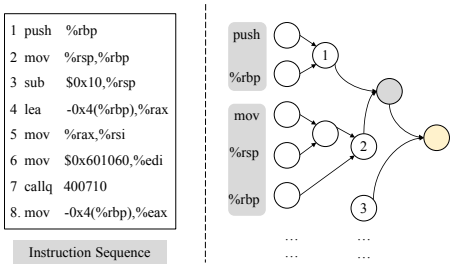

Figure 10. An Illustration of RAE.

The parameters $\{U, V, W\}$ are trained using back-propagation 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 embedding $U \in \mathbb{R}^{nm}$ 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 10, 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 [35] 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 10 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 [12, 50] for natural language processing and train a multi-class classifier using CNN for function mapping. Note that another line of work, such as tainting [30, 47], 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*. The process begins at the function entry point, traverse subsequent basic blocks in the function body, and select each branch with equal probability when necessary until a

function exit point is reached. All instructions visited along the process are collected to construct an (sample) execution path, and the function name is assigned as its class label. We also record the branch decisions in the *random walk* approach to both eliminate duplicated paths and improve code coverage. Once trained CNN classifier enables us to automatically map the execution path from dynamic instructions trace to function labels, completing **AMASS**'s feature identification.

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. Methods for 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.

Retaining features. We consider the case where a user only needs a set of features $\hat{\mathcal{F}}$. To deliver the tailored program, we execute the original program with available test-cases to generate dynamic traces that reach each feature in $\hat{\mathcal{F}}$. Through feature identification, we identify the set of constituent functions F_i of each feature i and derive the superset $\hat{F} = \cup F_i$, which is the set of functions we need to retain in the customized binary. Due to possible missing constituent functions during feature identification and deep learning, the set $\hat{F}$ may not contain all necessary functions to execute the resulting binary. We apply static CFG analysis to find and add any required functions that make $\hat{F}$ complete. In particular, each function in $\hat{F}$ will be mapped to the pre-built static CFG and the reachability analysis in section 4.2 will ensure that each mapped node in the CFG can be reachable from the function start.

Removing features. We consider now removing a set of features $\hat{\mathcal{F}}$ from a given program binary,

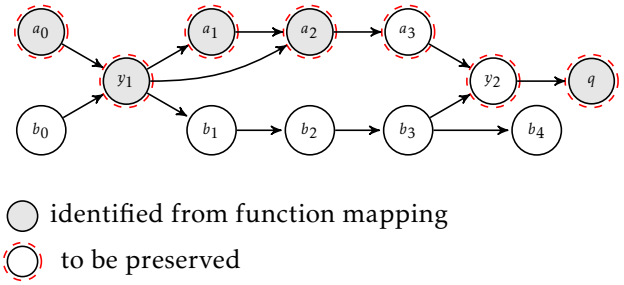

Figure 11. Reachability analysis on LibreOffice: retaining feature

which is useful when a user deems these features either unnecessary or vulnerable. To this end, we again execute the binary with test-cases to reach each unwanted feature in $\hat{\mathcal{F}}$. Then, after identifying each F_i from dynamic trace, the superset of constituent functions $\hat{F} = \cup_i F_i$ that correspond to the unwanted features can be obtained. However, for feature removal, we cannot simply eliminate all functions in $\hat{F}$ from the binary, due to the existence of shared functions with other (desired) features, which are required for the soundness of the customized program. Let $\hat{S}$ be the set of functions/basic blocks shared by other features (which can be found using the constituent functions of other features). **AMASS** will only remove functions/basic blocks in $\hat{F} - \hat{S}$, which are only needed for the operation of the unwanted features.

Tailoring via set operations. When the target features' constituent functions $\hat{F}$ are not directly identifiable, **AMASS** 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. **AMASS** 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. Reachability Analysis

A program's CFG can be represented as a directed graph, $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where the node set $\mathcal{V} = \{v_1, v_2, ..., v_m\}$

Algorithm 1 Reachability Analysis

Static CFG: $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$
Initial feature set:$\mathscr{F} = \{F_0, F_1, ..., F_n\}$
$\mathscr{F}'$=final feature set
Initialization:$\mathscr{F}' = \mathscr{F}$
for F_k in $\mathscr{F}$ **do**
  Find V^f: $V^f \supset F_k$ && $V^f \in \mathscr{V}$
  Find $\mathscr{T} = \{T_{head}, T_1, ..., T_m, T_{tail}\}$: $\mathscr{T}$ is the control flow path that contains F_k
  if $(V^f.entry \geq 2 \,||\, V^f.exit \geq 2)$ && $\mathscr{F}$ is for feature removal **then**
    $\mathscr{F}' = \mathscr{F}' - \{V^f\}$
  end if
  if $\exists \mathscr{T}$ **then**
    $\mathscr{F}' = \mathscr{F}' \cup \mathscr{T}$
  end if
end for

represents basic blocks and edge set $\mathscr{E} = \{e_1, e_2, ..., e_n\}$ represents control flows among basic blocks. The customized program can be viewed as a subgraph $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$, for $\mathscr{V} \subseteq \mathscr{V}$ and $\mathscr{E} \subseteq \mathscr{E}$. Ideally, for a given set of desired features, **AMASS**'s feature identification and tailoring modules should obtain their feature-constituent functions $\hat{F}$ that meet the following two requirements: (i) All functions in $\hat{F}$ should belong to desired features; (ii) The functions in $\hat{F}$ together can ensure that the desired features are functional, i.e., the customized binary can be executed with inputs that can reach the desired features. However, these may always hold because deep learning-based algorithm cannot guarantee to always produce the correct function mapping and feature identification. Necessary functions for the soundness of the customized binary may be missing, causing the program to crash and unable to execute the desired program features. We propose a CFG-based reachability analysis to tackle the issue. We design an algorithm as shown in Algorithm 1 to rectify possible missing functions and ensure the soundness of customized binary by expanding the identified feature functions. The basic idea is to connect the missing links in the CFG and preserve the shared code segments. As Algorithm 1 shows, we apply the following methods in the CFG: (i) If the basic block can jump to multiple targets ($V^f.exit \geq 2$), or multiple basic blocks can jump to this basic block ($V^f.entry \geq 2$), then this basic block is considered as a shared code segment. Hence, this basic block will not be removed in any case.

In the example shown in figure 11 from LibreOffice, where two features are intertwined. The circles represent the basic blocks and gray circles are those identified by feature identification module. The feature $\mathscr{F} = \{a_0, y_1, a_1, a_2, q\}$ is the desired feature that the user wants to keep. Without reachability analysis, a_3 and y_2 won't be kept in the customized binary since they are not identified by deep learning mapping. According to Algorithm 1, the identified basic blocks that belong to the same function will be connected by adding the missing nodes along with the control flow. We define $\mathscr{T}$ as the control flow path that resides within the scope of one function and contains all elements in $\mathscr{F}$. In this case, all the basic blocks in $\mathscr{T} = \{y_1, a_1, a_2, a_3, y_2\}$ should be included in $\mathscr{F}'$ even if a_3 and y_2 are not discovered by feature identification module. The nodes with red dashed circles are the final elements in the feature set to be customized.

4.3. Binary Rewriting

We use feature tailoring and reachability analysis 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, **AMASS** 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 of abstracting the program's basic blocks in the form of CFG. To remove the features in the programs, there are two steps in **AMASS**First, **AMASS** removes the functions that should not be called. The call site of the eliminated functions will be replaced to redirect the program to the exit point. Second, for those functions cannot be removed from the first step (e.g., For indirect function calls, the address of the callee function that 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 [48] by **AMASS** to validate the effectiveness and correctness of feature tailoring. Specifically, the fuzzing engine generates two sets of test cases: (1) F_1 that invokes the desired features in customized program; (2) F_2 that involves at least one of the eliminated features. In particular, **AMASS** usesF_1 to confirm the integrity of necessary program functionalities, while F_2 helps verify the successful removal and handling of eliminated features.

5. Evaluation

In this section, we evaluate the performance of each module in **AMASS** and the impact of feature customization.

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

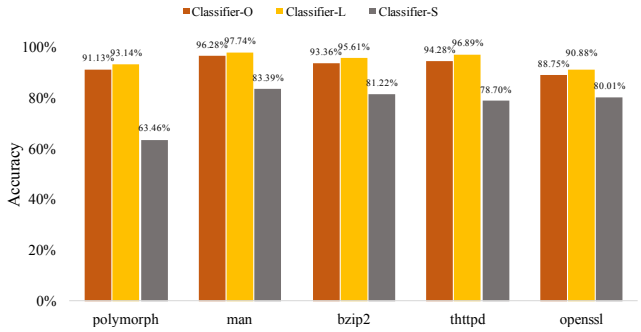

Figure 12. Accuracy of function mapping during feature identification

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* [16], 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 **AMASS** 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

Benchmark	#Functions	Vocabulary Size	#Training Execution paths	#Tokens
polymorph	23	201	10,806	460,248
man	77	1,198	346,570	86,008,653
bzip2	79	1,251	135,738	54,809,155
thttpd	129	1,838	189,855	54,162,084
OpenSSL	4,023	10,582	586,817	137,293,197

Table 1. Benchmark Profiles

(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 12. 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. Feature Combinations

In this section, we evaluate the number of customized programs after feature tailoring. When the features are identified by customers, we can create multiple customized binaries containing different feature combinations. Table 2 shows the number of selected features for each benchmark and the number of customized programs we are able to create. Our approach is able to produce numbers of customized programs to match the customer needs while minimizing unwanted exploitation of the applications features. The number of customized programs is calculated after feature tailoring. Since each feature can contain both unique functions and shared functions as mentioned in Section 2, there are some scenarios that several features cannot be

Bench.	*polymorph*	*bzip2*	*hmmer*	*thttpd*	*links*	*LibreOffice*	*OpenSSL*
Program Size (LoC)	404	5,904	20,721	7,956	178,441	4,485,797	305,279
#Selected Features	6	8	7	12	12	10	14
#Combinations	64	96	48	3,072	6,144	768	4,096

Table 2. Number of identified features and customized programs by **AMASS**.

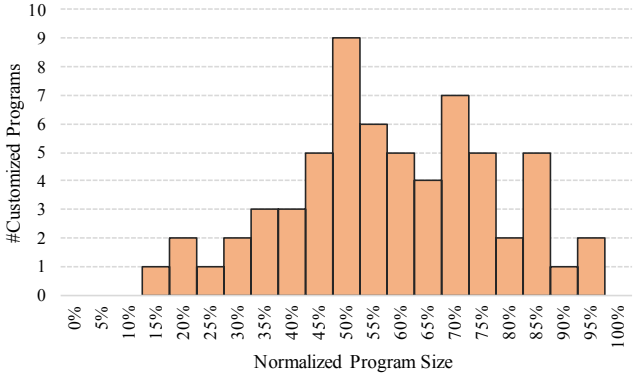

Figure 13. Number of customized program versions and their sizes normalized to original program (polymorph benchmark)

customized separately. For example, the 6 features we selected in polymorph are all independent to each other and totally separable. Hence we are able to create 2^6 customized programs. However, in LibreOffice, there are 2 selected features: print files and print files to a specific printer, they both execute print feature and have shared functions. They either can be removed or retained together. Thus, **AMASS** cannot create a customized version of LibreOffice with arbitrarily feature combinations.

We also evaluate the size of customized program variations. We pick one example program polymorph(a Win32 to Unix filename converter) to present the result. We identified 6 features in polymorph as: convert file, convert all hidden files, clean files, help/usage, trace file path, and program version. Figure 13 shows the program size distribution in terms of normalized program size in polymorph. As we can see, we generate various combinations of customized programs that contain *just-enough* software features to support specific use-cases and can significantly reduce the program size up to 85%.

5.4. 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 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 3. We evaluate the security impact of **AMASS** using the ratio of CVEs that can be removed by feature customization.

6. Related Work

Binary reuse: Binary reuse has been addressed by several works [29, 42, 49]. The reuse of binary code, different from source code, carries great difficulty. Methods proposed in [5] identify self-contained code fragment from binary with the help of both static disassembling and dynamic execution monitoring. There are also research works that focus on reconstructing program binary from dynamic traces, by utilizing instruction trace and memory dump [14, 43]. However, neither of the above two methods fits in the context of program feature customization due to limited degree of flexible modification, as it only focuses on segment reuse and high level assembly code. Moreover, even if the code can be customized, the newly compiled binary may fail to fulfill the purpose of feature customization.

Code analysis and De-bloating: Several prior works have proposed program customization frameworks only based one methods like de-bloating [9], cross-host tainting [6] and so on. In terms of binary reuse, it has been studied by several works [38, 41–43]. 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

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

Table 3. Impact on Application and Communication security

analysis [31, 32, 45]. 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 [44, 46] and SARRE [15] propose frameworks combining statistical and formal analysis for vulnerable path discovery. SIMBER [40] 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 **AMASS**, 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 **AMASS**, 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 **AMASS** 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 [28], 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 [25] for further performance improvements. Fundamentally, deep learning approach cannot guarantee zero false positives. In this case, we can provide feedback to the training phase of deep learning module as soon as we observe false positives. This will be helpful to improve the accuracy of our function mapping module. 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] Caballero, J., Johnson, N.M., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. Tech. rep. (2009)
[6] Chen, Y., Sun, S., Lan, T., Venkataramani, G.: Toss: Tailoring online server systems through binary feature customization. In: FEAST workshop (2018)
[7] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., et al.: The matter of heartbleed. In: Internet Measurement Conference (2014)
[8] Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. ACM SIGARCH Computer Architecture News (2005)
[9] Jiang, Y., Wu, D., Liu, P.: Jred: Program customization and bloatware mitigation based on static analysis. In: IEEE Computer Software and Applications Conference (2016)
[10] Jiang, Y., Zhang, C., Wu, D., Liu, P.: Feature-based software customization: Preliminary analysis, formalization, and methods. In: IEEE High Assurance Systems Engineering (HASE)
[11] Jiang, Y., Zhang, C., Wu, D., Liu, P.: Feature-based software customization: Preliminary analysis, formalization, and methods. In: High Assurance Systems Engineering (2016)

[12] Kim, Y.: Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882 (2014)
[13] Korkin, I., Tanda, S.: Detect kernel-mode rootkits via real time logging & controlling memory access. arXiv preprint arXiv:1705.06784 (2017)
[14] Kwon, Y., Wang, W., Zheng, Y., Zhang, X., Xu, D.: Cpr: cross platform binary code reuse via platform independent trace program. In: ACM International Symposium on Software Testing and Analysis (2017)
[15] 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)
[16] 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)
[17] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Acm sigplan notices (2005)
[18] 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)
[19] Mikolov, T., Kombrink, S., Deoras, A., Burget, L., Cernocky, J.: Rnnlm-recurrent neural network language modeling toolkit. In: ASRU Workshop (2011)
[20] 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)
[21] Mort, S.: Cve-2017-5638: Anatomy of the apache struts vulnerability (2017), https://blog.blackducksoftware.com/cve-2017-5638-anatomy-apache-struts-vulnerability
[22] Murphy, G.C., Lai, A., Walker, R.J., Robillard, M.P.: Separating features in source code: An exploratory study. In: Software Engineering
[23] Sanchez, A.: Personal banking apps leak info through phone (2014), http://blog.ioactive.com/2014/01/personal-banking-apps-leak-info-through.html
[24] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: Security and Privacy (2016)
[25] 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)
[26] Snyder, P., Ansari, L., Taylor, C., Kanich, C.: Browser feature usage on the modern web. In: Internet Measurement Conference (2016)
[27] Source, O.: Libreoffice
[28] 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)
[29] van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: Security and Privacy (2016)
[30] 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)
[31] 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)
[32] 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)
[33] Viega, J., Messier, M., Chandra, P.: Network Security with OpenSSL: Cryptography for Secure Communications. " O'Reilly Media, Inc." (2002)
[34] Wang, Y., Wu, Z., Wei, Q., Wang, Q.: Neufuzz: Efficient fuzzing with deep neural network. IEEE Access (2019)
[35] 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)
[36] Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., Sevitsky, G.: Finding low-utility data structures. ACM Sigplan Notices (2010)
[37] Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: FSE/SDP workshop on Future of software engineering research (2010)
[38] Xue, H., Chen, Y., Venkataramani, G., Lan, T.: Hecate: Automated customization of program and communication features to reduce attack surfaces. In: International Conference on Security and Privacy in Communication Systems (2019)
[39] 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)
[40] 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)
[41] Xue, H., Sun, S., Venkataramani, G., Lan, T.: Machine learning-based analysis of program binaries: A comprehensive study. IEEE Access (2019)
[42] 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)
[43] Xue, H., Venkataramani, G., Lan, T.: Clone-slicer: Detecting domain specific binary code clones through program slicing. In: FEAST workshop. ACM (2018)
[44] Xue, H., Venkataramani, G., Lan, T.: Twin-finder: Integrated reasoning engine for pointer-related code clone detection. arXiv preprint arXiv:1911.00561 (2019)

[45] 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
[46] 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)
[47] Yao, F., Venkataramani, G., Doroslovački, M.: Covert timing channels exploiting non-uniform memory access based architectures. In: Great Lakes Symposium on VLSI. ACM (2017)
[48] Zalewski, M.: American fuzzy lop (2007)
[49] Zeng, J., Fu, Y., Miller, K.A., Lin, Z., Zhang, X., Xu, D.: Obfuscation resilient binary code reuse through trace-oriented programming. In: ACM conference on Computer & communications security (2013)
[50] 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)