

# POSTER: MORPH: Enhancing System Security through Interactive Customization of Application and Communication Protocol Features

Hongfa Xue

The George Washington University  
Washington, DC, USA  
hongfaxue@gwu.edu

Yurong Chen

The George Washington University  
Washington, DC, USA  
gabrielchen@gwu.edu

Guru Venkataramani

The George Washington University  
Washington, DC, USA  
guru@gwu.edu

Tian Lan

The George Washington University  
Washington, DC, USA  
tlan@gwu.edu

Guang Jin

Intelligent Automation Inc.  
Rockville, Maryland, USA  
gjin@i-a-i.com

Jason Li

Intelligent Automation Inc.  
Rockville, Maryland, USA  
jli@i-a-i.com

## ABSTRACT

The ongoing expansion and addition of new features in software development bring inefficiency and vulnerabilities into programs, resulting in an increased attack surface with higher possibility of exploitation. Creating customized software systems that contain *just-enough* features and yet satisfy specific user needs is currently an extremely slow, build-to-order process. In this paper, we propose MORPH, an Interactive Program Feature Customization framework to provide broad capabilities for automated program feature identification and feature customization. Our preliminary results show that MORPH can identify program features at an average accuracy of 92.7% and swiftly generate variations of self-contained, customized programs in an unsupervised fashion.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software security engineering**;

## KEYWORDS

Program customization; Debloating; Machine learning; Binary analysis

### ACM Reference Format:

Hongfa Xue, Yurong Chen, Guru Venkataramani, Tian Lan, Guang Jin, and Jason Li. 2018. POSTER: MORPH: Enhancing System Security, through Interactive Customization of Application, and Communication Protocol Features. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, October 15–19, 2018, Toronto, ON, Canada. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3243734.3278518>

## 1 INTRODUCTION

The rapid inflation of software features results in not only larger software installation footprint, but also an increased attack surface with higher possibility of vulnerabilities. This is especially true in implementations of communication protocols. For example, Simple Network Management Protocol (SNMP) is designed for managing network devices, and Net-SNMP is a widely used SNMP

implementation for both Linux and Windows. CVE-2014-3565 [2] vulnerability discovered that an attacker can craft an SNMP trap message to cause a Denial of Service (DoS) attack. While the trap communication in the SNMP is used for network devices to report events, it is never invoked under most practical scenarios.

In this paper, we propose MORPH, an Interactive Program Feature Customization framework to provide broad capabilities for (i) identifying features from various programs and communication protocols and (ii) tailoring/removing (unnecessary) program features and validating the correctness and compatibility of the program customization. MORPH enables two different solutions to identify program features automatically. First, when feature seed information (such as functions and basic blocks that are unique to target feature) are provided, starting from these seed information, MORPH traverses the program Control Flow Graph (CFG) and finds the features of interest. Second, when dynamic traces (that execute different combinations of program features, e.g., as invoked by test-cases) are available, MORPH leverages deep learning to traverse basic blocks in the traces and map them to features. Next, by converting a program into its equivalent Intermediate Representation (IR) of Low Level Virtual Machine (LLVM), MORPH can modify the program for different needs. The modified program can be compiled back into an executable binary for further evaluation and validation. Further, MORPH will enable interactive feature customization process and needs minimal user's involvement. In particular, it makes use of fuzzing technique to automatically test the customized program while focusing on the removed features. The test results are reported back to users to indicate potential issues of improperly modified program.

To evaluate the effectiveness of MORPH, we provide a preliminary implementation and select several real-world applications/protocols, including `httplib`, `LibreOffice` and `Openssl`. We evaluate the performance of MORPH's deep learning algorithm for automated feature identification from dynamic execution trace. MORPH achieves an average 92.76% accuracy for feature identification and mapping to binary code. These preliminary results demonstrate MORPH's ability to swiftly create a large variation of self-contained, customized programs in an unsupervised fashion.

## 2 PROBLEM STATEMENT

### 2.1 Problem Statement and Challenges

In general, program customization involves two tasks: (i) identifying program features from a binary, and (ii) rewriting the binary, in accordance with user needs, to create customized programs. The goal of feature customization creates customized software systems

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS '18, October 15–19, 2018, Toronto, ON, Canada

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5693-0/18/10.

<https://doi.org/10.1145/3243734.3278518>

that contain *just-enough* features to support specific use-cases and can considerably reduce the software’s attack surface, as unwanted features are eliminated even before zero-day exploits.

## 2.2 Definitions

**DEFINITION 1. Feature.** A program feature is defined as a set of basic blocks – denoted by  $F_i = \{b_i^1, b_i^2, \dots, b_i^n\} \subseteq \mathcal{F}$  – 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.

**DEFINITION 2. Control Flow Graph (CFG).** A CFG,  $G = (B, J)$ , is directed graph comprising a node set,  $B$ , and an edge set,  $J$ . In a CFG, a node,  $b_i \in B$ , indicates a basic block which represents a sequential execution of instructions without any jump instructions. In a CFG, an edge,  $j_i \in J$ , connecting two basic blocks  $b_n$  and  $b_m$  indicates a control flow change,  $j_i = (b_n, b_m)$ , meaning jump from  $b_n$  to  $b_m$ .

Identifying program features can be a very challenging problem, because features often traverse multiple basic blocks/functions across different regions of the binary, with the control flow not possible to resolve statically.

## 3 SYSTEM DESIGN

Figure 1 shows a schematic of MORPH, that takes a program binary as input, and generates the program’s CFG in the IR of LLVM. Feature customization in MORPH are done in an interactive and closed-loop fashion. First, program features are automatically identified from seed information or dynamic trace. Second, after reviewing the identified features, a user guides MORPH on how to automatically rewrite the program and remove selected features. Third, MORPH generates a customized/feature-removed program based on user’s directions, and feeds the program to an automatic testing engine focusing on the removed feature. Then, the testing engine evaluates the program and provides test results back to the MORPH user. Based on the testing report, an MORPH user decides whether the customized program meets the requirement and if further iterations are needed.

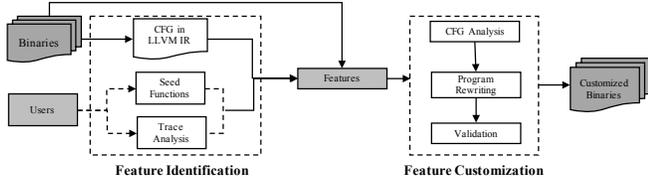


Figure 1: MORPH framework Overview

### 3.1 CFG analysis

MORPH leverages the LLVM to facilitate the CFG generation and additional analysis. MORPH produces the LLVM IR as the CFG representation for the selected program implementation. Based on the generated CFG, LLVM provides necessary utilities to traverse, analyze and modify CFGs. MORPH will support customizing a program base on its binary format. The output of CFG and Instruction trace analysis allows MORPH to perform further analysis, identify and remove program features.

### 3.2 Feature Identification

**3.2.1 Feature identification by seed information.** MORPH can take seed basic blocks and functions as inputs that define unique feature operations, capabilities, or system service accesses, traverse the

CFG, and identify all constituent basic blocks for each feature on the CFG. With feature seed information available, we have evaluated the ability of MORPH to identify and isolate different program features in our prior work, DamGate [4] to show that the desired level of feature isolation can be achieved with low runtime overheads.

**3.2.2 Feature identification by trace analysis.** Using test-cases available to invoke various program features, we can obtain dynamic trace that contains the execution of different combinations of program features, extract the related execution paths, and identify the constituent basic blocks for each feature (or feature combination) in the binary. To this end, we (i) apply trace splicing to extract dynamic execution paths (of invoked features) from the dynamic trace, and (ii) map them to basic blocks in the binary code, to identify program features through their constituent basic blocks. This is a challenging problem because some code blocks in the application binary can have multiple entry points, and hence, the dynamic traces could differ for the same code block. Finally, we consider this mapping problem as a multi-class classification problem, where each function in the binary is considered as a class label, the function’s execution path and basic blocks as samples of the class, and an execution path extracted from dynamic instruction trace as the testing sample. Thus, we can 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. Note that another line of work, such as tainting [9, 10], can be used for feature identification. We consider this as future work.

### 3.3 Feature Customization

**3.3.1 Interactively remove features .** MORPH provides an interactive process for users to guide in the program customization. An entry block within a feature has no incoming edge, while an exit block has no outgoing edge. A feature may have multiple entry blocks and multiple exit blocks. Entry blocks defines the entry points for a feature, while exit blocks defines potential return values of the feature. To modify a binary executable and remove a feature, MORPH has two options to remove features. (a) **Rewriting exit blocks:** only modifies exit blocks can leave the original CFG mostly intact, and thus may leave the vulnerability unchanged in the modified program. (b) **Rewriting entry blocks:** directly redefines the internal logics of certain program APIs for the identified feature. Modifying entry blocks can easily orphan other blocks (i.e., making these block inaccessible either internally or externally). The orphan blocks can then be clearly marked and removed. Since the basic blocks in an identified feature are connected, these blocks can be topologically sorted. MORPH will provide necessary interfaces for users to specify how to modify an entry or exit block.

**3.3.2 Validate customized programs.** For the features preserved in the customized programs, MORPH will not change the execution trace of such features. After binary rewriting, standard program fuzzing techniques [14] can be employed by MORPH to validate if a customized program can still interact with an unmodified application, and won’t crash even if the identified feature has been removed. MORPH uses fuzzing to confirm the integrity of program functionalities after customization.

## 4 PRELIMINARY RESULTS

### 4.1 Experiment Setup and Data Collection

We implemented each of the module in our system to collect early experiment results. Our experiments are conducted on a 2.80 GHz Intel Xeon(R) CPU E5-2680 20-core server with 16 GByte of main



Figure 2: Accuracy of function mapping

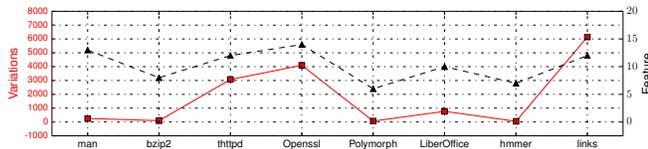


Figure 3: Number of features and customized variations generated by MORPH, where the black dot line represents the number of features for each benchmark and red line represents the number of customized programs correspondingly

memory. We select different sets of real world applications: (i) SPEC 2006 suite [1], bzip2 and hmmer; Bugbench suite *bugbench* [6], polymorph and man and (ii) Interactive applications including a light-weight web server tthttpd, version beta 2.23, an open source office suite LibreOffice and a web browser, links. (iii) A TLS and SSL protocol, Openssl. In our trace analysis module, we collect static execution paths using Depth First Search as training dataset and dynamic execution paths as testing dataset for evaluating the accuracy of the pre-trained models. We chose the hidden node size as 500 in RNN and 200 maximum iterations for RAE.

#### 4.2 Accuracy of function mapping

Figure 2 presents the accuracy of function mapping through trace analysis in Feature identification. MORPH achieves an overall average accuracy of 92.76%, with the highest up to 96.28% in *man* from *bugbench*. We note that the mapping accuracy of larger programs, such as bzip2 and tthttpd, is higher than that of smaller programs like polymorph due to larger execution traces available for training the CNN classifiers. For the applications with more functions, such as Openssl, the overall accuracy can be as low as 88.75% since there are more classes for classification. Further improving the performances via deep learning or by introducing formal analysis provides directions for interesting future work.

#### 4.3 Feature Customization

Figure 3 shows the number of selected features for each benchmark and the number of customized programs we are able to create. When the features are identified by users, MORPH produces multiple customized binaries containing different feature combinations. The number of customized programs is calculated after interactive feature removal. Since each feature can contain both unique and shared functions, there are some scenarios that multiple features are tightly coupled and cannot be customized separately.

### 5 RELATED WORK

Program de-bloating removes redundant code from programs. Most prior works focus on source code. For example, Jred [5] proposes a method to remove unused methods in JAVA program, which utilizes call graph analysis and operates at IR level. Different from program

de-bloating, binary reuse is aimed at reconstruct program binary from execution trace for further analysis. Moreover, none the above works are aimed at program feature customization.

Learning-based approaches have been proposed for both static and dynamic program analysis. StatSym [13] combines symbolic execution and statistics analysis for vulnerability discovery. SIMBER [11] proposes a statistical inference framework to eliminate redundant bound checks and improve the performance of applications without sacrificing security. Clone-Hunter [12] also takes advantage of binary code clone detection for accelerated bound check removal. In MORPH, deep learning-based methods are used for trace analysis. Program bug analysis [8] and tuning [3, 7] have been used alternatively for program customization.

### 6 CONCLUSION AND FUTURE WORK

In this paper, we propose MORPH, an Interactive Program Feature Customization framework to provide broad capabilities for automated program feature identification and feature customization. MORPH enables users with two solutions for feature identification and interactively remove program features with validation. In our preliminary implementation and evaluation of MORPH, we chose real-world applications and our preliminary results shows that MORPH is able to identify features at an average accuracy of 92.7% and generate a large variation of self-contained customized programs. As future work, we consider automation techniques for fuzzing techniques.

### 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 are those of the authors, and do not necessarily reflect those of ONR.

### REFERENCES

- [1] 2006. SPEC CPU 2006. <https://www.spec.org/cpu2006/>.
- [2] 2014. CVE-2014-3565. <https://www.cvedetails.com/cve/CVE-2014-3565/>.
- [3] J. Chen, G. Venkataramani, and H. H. Huang. 2012. RePRAM: Re-cycling PRAM faulty blocks for extended lifetime. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.
- [4] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation*.
- [5] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *40th Annual Computer Software and Applications Conference*.
- [6] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*.
- [7] Jungju Oh, Christopher J. Hughes, Guru Venkataramani, and Milos Prvulovic. 2011. LIME: A Framework for Debugging Load Imbalance in Multi-threaded Execution. In *Intl. Conference on Software Engineering*.
- [8] Jianli Shen, Guru Venkataramani, and Milos Prvulovic. 2006. Tradeoffs in fine-grained heap memory protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM.
- [9] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2009. Flexitaint: A programmable accelerator for dynamic taint propagation. In *IEEE 14th International Symposium on High Performance Computer Architecture*.
- [10] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2009. MemTracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)* (2009).
- [11] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. SIMBER: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*.
- [12] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-hunter: accelerated bound checks elimination via binary code clone detection. In *2nd ACM SIGPLAN Intl. Workshop on Machine Learning and Programming Languages*.
- [13] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. 2017. StatSym: vulnerable path discovery through statistics-guided symbolic execution. In *International Conference on Dependable Systems and Networks*.
- [14] Michal Zalewski. 2007. American Fuzzy Lop.