Debloating Feature-Rich Closed-Source Windows Software

Zhen Huang
School of Computing
DePaul University
Chicago, USA
zhen.huang@depaul.edu

***Abstract*—Feature-rich software programs typically provide many configuration options for users to enable and disable features, or tune feature behaviors. Given the values of configuration options, certain code blocks in a program will become redundant code and never be used. However, the redundant code is still present in the program and thus unnecessarily increases a program's attack surface by allowing attackers to use it as return-oriented programming (ROP) gadgets. Existing code debloating techniques have several limitations: not targeting this type of redundant code, requiring access to program source code or user-provided test inputs.**

In this paper, we propose a practical code debloating approach, called BinDebloat, to address these limitations. BinDebloat identifies and removes redundant code caused by configuration option values. It does not require user-provided test inputs, or support from program developers, and is designed to work on closed-source programs. It uses static program analysis to identify code blocks that are control-dependent on configuration option values. Given a set of configuration option values, it automatically determines which of such code blocks become redundant and uses static binary rewriting to neutralize these code blocks so that they are removed from the attack surface. We evaluated BinDebloat on closed-source Windows programs and the results show that BinDebloat can effectively reduce a program's attack surface.

I. Introduction

Many software programs offer configuration-based features to meet the diverse requirements of different users. A user can conveniently enable or disable a feature by changing the corresponding configuration options. As user requirements become more and more complicated, the number of features and their corresponding configuration options increase dramatically. Popular programs such as MS Office, Adobe Acrobat Reader, Apache HTTP server, and MySQL typically have hundreds of configuration options [1], [2].

While a feature-rich program provides a large number of configuration options, each individual user usually uses only a small subset of them. Prior study has shown that below 17% of the configuration options are used by the majority of users [2]. However, the code exclusively used by the unused features still remain in the program and leaves the program a bloated attack surface. Despite decades of effort in addressing vulnerabilities [3]–[11], the unused code can contain vulnerabilities and lead to severe security issues, such as being exploited by return-oriented programming (ROP) attacks [12] and data-only attacks [13].

To remedy such issues, code debloating techniques have been proposed to reduce the attack surface of the programs or the underlying API libraries or operating systems. These techniques can be roughly categorized into three directions. One direction is to remove the extraneous code from the programs [14] so that the attacks can no longer exploit unused code in programs. Another direction is to remove the code of API functions not used by programs [15]. This way attackers cannot take advantage of unused API functions. The other direction is to specialize API functions [16], or customize OS kernels [17]. By restricting the way to use API functions or OS system calls, illegitimate use of them will be prevented.

However, these techniques have several limitations. First, the majority of them do not target configuration-based features. Some of them require developers to annotate feature-specific code [18]. Some ask developers to provide one or more seed functions for identifying features [19]. Second, few of the techniques targeting configuration-based features work on closed-source software. The vast majority of them need access to program source code [20], [21], which prevents them from being applicable to proprietary software or legacy software, which often does not have source code. Third, many of them require users to provide a set of test inputs to identify the mapping of configuration to code [14], [22], which is a non-trivial task for end-users and even developers.

The goal of this work, called BinDebloat, is to overcome these limitations. We focus on programs that provide configuration-based features and develop static program analysis techniques to map configuration options to the binary code of a program that implements these features, which require neither source code nor test inputs. For each configuration option, we identify the code that will be conditionally executed based on the value of the configuration option. Based on the mapping, our work reduces the attack surface by removing from the program the binary code that will never be executed for the given setting of configuration options.

We choose to use configuration options to identify program features because 1) programs commonly provide configuration options for users to enable or disable specific program features, and 2) programs usually use standard configuration storage to store and access configuration options, such as using Windows registry or text-based configuration files.

We developed a prototype of BinDebloat that automatically

identifies in a closed-source program the redundant binary code, for a set of values of configuration options, and uses binary rewriting to neutralize the redundant code. As most Windows programs use Windows registry for storing configuration options, BinDebloat targets Windows programs that use Windows registry. It leverages static program analysis to find the mapping of configuration options and code. To neutralize redundant code, it replaces redundant code with instructions that will trigger software interrupts.

This work makes the following major contributions:

- We describe the design and implementation of BinDebloat, an approach to debloat redundant code in closed-source programs. BinDebloat automatically identifies and removes code blocks redundant for a set of configuration option values used by a program. It does not requires support from program developers or test inputs.
- We evaluate BinDebloat on two popular real-world closed-source Windows programs, 7zip file archiver and Adobe Acrobat DC PDF reader. The evaluation illustrate the potential of BinDebloat to effectively reduce the attack surface of closed-source programs.

II. Related Work

Program Debloating. Code debloating removes redundant code from a program. The redundant code is typically defined as the code exclusively used by unneeded program features. When a feature is not used by a user, its corresponding code is considered redundant. One major challenge is how to define and identify program features.

Some tools require considerable effort from program developers or users to identify program features [18], [23]. CARVE expects program developers to identify features with program annotations [23]. The annotations map each feature to its corresponding source code by marking the beginning and the end of the code. To remove the redundant code for an unused feature, CARVE deletes the code. Its shortcoming is that annotating program code can be tedious and error-prone.

Based on the idea of delta debugging [24], CHISEL splits program code into multiple partitions and iteratively searches for a partition that can meet a user's requirements [18]. It relies on a user-provided property test function to identify features needed by a user. For each candidate partition, it runs the partition against the property test function to determine whether the partition is the desired one.

A few tools use configuration options to identify program features for debloating [20], [21]. TRIMMER reduces program code size by transforming program source code mainly with constant propagation based on the constant values of a given set of configuration options [20]. To delete redundant code, it uses loop unrolling in conjunction with constant propagation to customize the code for the values of the configuration options.

API Specialization. While code debloating focuses on the code of a target program, API specialization focuses on the API functions and libraries used by a target program. Most of the API specialization tools either create specialized versions of API functions or remove unnecessary API functions.

Piece-wise compiler [15] analyzes the source code of a program to find out which API functions are called by the program, and removes the API functions that will never be called by the program. C2C [21] prevents unnecessary calls to API functions from a program under a particular set of configuration option values. It identifies the set of system calls conditionally used by a program, and uses Seccomp BPF to restrict the system calls the program can use at runtime. While piece-wise compiler and C2C require access to program source code, Nibbler [25] works on binary code. It creates special versions of libraries for the program by statically removing the unneeded API functions from the libraries.

III. Motivation

Code bloating, also called feature creep, is common in feature-rich software programs because many features are not actually used by the vast majority of users [2]. Retaining the code implementing these unused features would unnecessarily increase the attack surface of programs.

Listing 1 illustrates an example of unused code in a proprietary PDF reader, Adobe Acrobat DC. The entry function of the PDF reader, `WinMain`, checks whether the user uses the feature for dynamic shared library optimization, which can be enabled or disabled by configuration option `bLTEnableDLLOptimization` in the Windows registry. By default, this option does not even exist, meaning that the option is disabled.

We can see that line 4 of the assembly code attempts to read the configuration option from Windows registry via a call to `RegGetValueW`, a Windows registry API function that retrieves a value of a configuration option. The result of the call determines whether to execute the instructions from memory address 0x14027309A to 0x1402731FE, which corresponds to 89 instructions.

If the configuration option does not exist, the jump at line 8 will be taken so that all the instructions from line 9 to line 97 will never be executed. If the configuration option exists but its `value` is zero, the jump at line 10 will be taken so that all the instructions from line 11 to line 97 will never be executed. As a result, these instruction do not need to be present in the program when the configuration option is absent or set to zero. Having these instructions in the program unnecessarily increases the attack surface and allows an attacker to use them as GOP gadgets for return-oriented-programming attacks [12].

State-of-art code debloating techniques reduce the attack surface of programs in various ways. However, they have several major limitations that make it difficult for users, particularly end users, to adopt them in practice. We believe that a practical code debloating solution should satisfy four requirements. First, it should target feature-rich programs that use configuration options to control whether to execute feature-specific code. Second, it should work on closed-source software (CSS) programs. Third, it should not require support from program developers, such as annotating code, providing

```
int WinMain(...)
; option: "bLTEnableDLLOptimization"
; aSoftwarePolicy: "SOFTWARE\Policies\Adobe\Adobe␣
    Acrobat\DC\FeatureLockdown"

; =============== Basic Block 1 ==============
 1 .text:14027306F lea r8, option
 2 .text:140273076 lea rdx, aSoftwarePolicy
 3 .text:14027307D mov rcx, hkey
 4 .text:140273084 call RegGetValueW
 5 .text:14027308A test eax, eax
 6 .text:14027308C jz 14027309A

; =============== Basic Block 2 ==============
 7 .text:14027308E ....
 8 .text:140273095 jmp 1402731FE

; =============== Basic Block 3 ==============
 9 .text:14027309A cmp value, 0
10 .text:1402730A1 jz 1402731FE

; =============== Basic Block 4 ==============
11 .text:1402730A7 ....
.................  ....
97 ..............  ....

; =============== Basic Block 5 ==============
98 .text:1402731FE ....
.................  ....
```

Listing 1: Example assembly code containing configuration-dependent code blocks, adopted from the entry function `WinMain` of Adobe Acrobat DC, a popular PDF readeer.

TABLE I: State-of-art code debloating techniques.

Technique	Config. Options	CSS	Not Req. Dev. Support	Not Req. Test Inputs
Slimium [26]	✗	✓	✗	✗
CARVE [23]	✗	✗	✗	✓
Razor [27]	✗	✓	✓	✗
Shredder [16]	✗	✓	✓	✓
Nibbler [25]	✗	✓	✓	✓
TOSS [14]	✗	✓	✓	✗
[22]	✓	✗	✗	✗
C2C [21]	✓	✗	✗	✓
TRIMMER [20]	✓	✗	✓	✓
BinDebloat	✓	✓	✓	✓

seed functions for identifying features, or manually mapping features to code. Lastly, it should not require users to provide test inputs.

We summarize the extent to which representative state-of-art code debloating techniques satisfy these four requirements in Table I. The majority of them do not identify bloated code using configuration options. None of the ones do so can work with closed-source software. Furthermore, many of them require either program developer support or test inputs. This work, BinDebloat, is the only one satisfying all the four requirements.

We introduce the following terms for defining the problem addressed by BinDebloat.

Configuration Option. A configuration option, also called a configuration setting, refers to a value that can be changed by program users to control program behaviors, such as enabling or disabling a program feature. It is usually represented a pair of key and value, with the key specifying the name of the configuration option. Configuration options are usually stored in the file system, in the form of OS-provided storage, such as Windows registry and Linux GSettings, or text-based configuration files. In this work, we focus on configuration options that are stored in Windows registry.

Configuration-dependent Code. We define configuration-dependent code as the program code blocks that will be executed *if and only if* the value of a configuration option satisfies a condition. If the value of the configuration option does not satisfy the condition, the code blocks will *never* be executed.

Redundant Code. Given the values of the set of configuration options used by a program, some of the program's configuration-dependent code will never be executed, regardless of user inputs. We refer to such code as redundant code.

The goal of BinDebloat is to identify and remove redundant code in the binary code of closed-source programs, based on a set of configuration option values.

IV. Design and Implementation

A. Overview

BinDebloat debloats code in a program that will *never* be executed under a given set of the program's configuration option values. We call such kind of code *redundant code*. BinDebloat takes the binary code of a program and the program's configuration option values as inputs, and generates a customized binary code that do not contain the code not needed for the set of configuration option values, as shown in Figure 1. It debloats code in three phases: *labeling configuration-dependent code*, *identifying redundant code*, and *removing redundant code*.

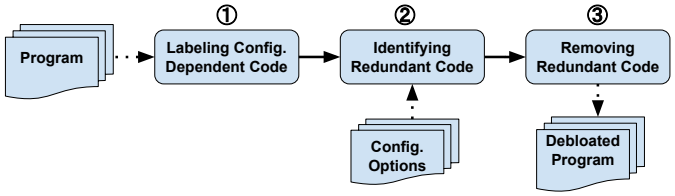


Fig. 1: BinDebloat Workflow: each rounded rectangle represents a step in BinDebloat; each circled number denotes a phase that consists of one or more steps underneath the circled number; dotted arrows denote input/output data, while solid arrows denote the order of steps.

In the first phase, it labels code blocks that are control-dependent on configuration options values. In other words, the values of configuration options determine whether these code blocks will ever be executed. This phase takes the binary code as input, and outputs a mapping from configuration options to code blocks that are control-dependent on configuration options. We call the mapping configuration-to-code mapping.

The second phase identifies the code blocks that need to be debloated for the given setting of configuration options.

Based on the configuration-to-code mapping, it reads the values of configuration options from the configuration storage, e.g. Windows registry or configuration files, and finds out which code blocks will never be executed for the given values of configuration options. It takes the configuration-to-code mapping and the configuration storage as input, and outputs a list of redundant code blocks.

The third phase removes the redundant code blocks from the binary code. It substitutes each instruction in these code blocks with an instruction that will trigger a software interrupt. It takes the binary code and the list of redundant code blocks as input, and outputs a debloated binary program.

For the example code in Listing 1, and a set of configuration option values in which the value of configuration option `bLTEnableDLLOptimization` is zero, BinDebloat will remove basic block 4 from the program. This is because that basic block 4 will never be executed if the configuration option is set to zero.

B. Labeling Configuration-Dependent Code

To be able to identify redundant code, BinDebloat needs to label configuration-dependent code. Each configuration-dependent code block needs to be associated with a predicate involving the configuration option on which the code blocks depend. The code blocks will never be executed if the predicate evaluates to `false`. This is performed in three steps: *finding access to configuration options*, *identifying configuration-dependent code blocks*, and *deriving constraints for configuration-dependent code blocks*.

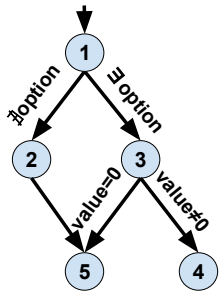
First, it disassembles the binary code of the target program into assembly instructions and builds control flow graphs (CFGs) of the instructions. It then searches for the program's access to configuration options. It finds instructions directly checking the existence of configuration options or reading configuration option values. Then it identifies instructions indirectly access configuration options via other functions.

Second, it builds the control dependency graphs (CDGs) of the instructions and follows the accesses to configuration options to identifies code blocks that are control-dependent on configuration options. It looks for branch instructions whose branches are dependent on the results of these accesses. Based on the CDGs, it identifies code blocks that are control-dependent on configuration options.

Lastly, it derives the constraints on which the execution of the configuration-dependent code blocks depend. The constraints for each basic block is a conjunction of predicates, each involves a configuration option. It labels each configuration-dependent code block with its constraints.

1) Finding Access to Configuration Options: BinDebloat uses `angr` to disassemble binary programs and build control flow graphs. Figure 2a shows the CFG for the code in Listing 1. The CFG consists of five basic blocks, in which both basic block 1 and basic block 3 end with a conditional branch.

To find direct access to configuration options, BinDebloat searches for instructions accessing configuration options. Because most programs use configuration storage provided by OSes to store configuration options, and call API functions to access configuration options, it looks for instructions calling configuration-related API functions. For example, it finds instructions calling Windows registry API functions, such as `RegGetValueW` and `RegQueryValueExW`, for programs using Windows registry to store configuration options.



(a) Control flow graph

BB2	∄option
BB3	∃option
BB4	∃option ∧ (value≠0)

(b) Basic block constraints

Fig. 2: Control flow graph and basic block constraints: each circle denotes a basic block while each arrow denotes a control flow.

Some programs have wrapper functions for accessing configuration options, and call these wrapper functions to indirectly access configuration options. Typically a non-wrapper function calls a wrapper function with the name of a configuration option as an argument, and the wrapper function passes the name argument to a configuration-related API function.

We note that a program may have multiple levels of wrapper functions. One wrapper function can call another wrapper function to access configuration options. As a result, BinDebloat follows the call graph to identify different levels of wrapper functions.

It focuses on wrapper functions that take the name of a configuration option as an argument. Starting from the functions directly accessing configuration options, it uses data flow analysis to link the arguments of these functions to the arguments used to call configuration-related API functions, and follows the call graph to find indirect access to configuration options.

2) Identifying Configuration-dependent Code Blocks: The instructions accessing configuration options can perform two types of accesses: retrieving the value of a configuration option and checking the existence of a configuration option. An instruction performing the former type of accesses typically stores the retrieved configuration option values into some memory locations, while an instruction performing the latter type of accesses typically stores the values indicating the existence of configuration options into CPU registers.

Based on the access type of these instructions, BinDebloat uses data flow analysis to find conditional branches whose conditions are derived from either the memory locations or the CPU registers used as the access results by the instructions. We call these conditional branches as configuration-related

branches.

BinDebloat then builds the CDGs of the program and iterates through the list of configuration-related branches to identify the list of code blocks that are control-dependent on these branches. These code blocks are deemed configuration-dependent code blocks.

3) Deriving Constraints for Configuration-dependent Code Blocks: A code block can have two different types of control dependencies on a configuration option: 1) value dependency or 2) existence dependency. A value dependency refers to the case when the dependency is on the configuration option value. An existence dependency refers to the case when the dependency is on whether a configuration option exists in the configuration settings. BinDebloat differentiates the two types of control dependencies based on the access type of the instructions accessing configuration options.

For each configuration-dependent code block, BinDebloat needs to derive the constraints that determine whether to execute the code block. The constraints is in the form of a conjunction of predicates, each of which involves a check on a configuration option.

To build a predicate, BinDebloat requires two types of information: the name of a configuration option and the check on the configuration option. BinDebloat finds the name of the accessed configuration options from the arguments used to call configuration-related API functions or wrapper functions.

If a code block and a configuration option has an existence dependency, BinDebloat only needs the name of the configuration option. If a code block and a configuration option has a value dependency, the check on the predicate must involve a comparison and a value. BinDebloat finds the type of the comparison and the value to be compared based on the semantics of the instruction retrieving a configuration option value to its corresponding configuration-related branch.

Using Figure 2a as an example, basic block 1 checks the existence of a configuration option. As a result, BinDebloat derives $\exists \texttt{option}$ as the constraints for basic block 3. Because basic block 3 checks whether a value is zero, BinDebloat derives $\exists \texttt{option} \wedge \texttt{value} \neq 0$ as the constraints for basic block 4, as shown in Figure 2b.

C. Identifying Redundant Code

Once constraints are derived for configuration-dependent code blocks, BinDebloat will be able to identify redundant code for any given set of configuration option values. It checks which code blocks' constraints cannot be satisfied based on the configuration option values and considers such code blocks as redundant code.

First, it builds a mapping from configuration options to predicates, which are associated with configuration-dependent code blocks. Second, for each configuration options in the mapping, it uses the configuration option values to determine which predicates cannot be satisfied. Last, it marks the code blocks corresponding to the unsatisfiable predicates as redundant code.

For programs using configuration storage provided by OSes, such as Windows registry, BinDebloat calls configuration-related API functions to retrieve the configuration option values. For programs using customized configuration storage, such as configuration files, BinDebloat calls user-provided functions retrieve the configuration option values.

D. Removing Redundant Code

BinDebloat removes identified redundant code to reduce the program's attack surface. One method to remove instructions from a binary program is to delete the instructions constituting redundant code from the binary program. However, deleting the instructions will require either recompiling the code [27] or complex binary rewriting, because the addresses for the instructions and data, following the deleted instructions will be changed. Besides that, a major disadvantage is that it may introduce new ROP gadgets [28].

Another method is to change each of these instructions into a NOP [14] or a software interrupt instruction [25]. Preserving the bytes corresponding to the "removed" instructions in a binary program will avoid the issue of changing the addresses of instruction or data following the "removed" instructions. By changing the instructions to NOP or software interrupt instruction, they cannot be used as any ROP gadgets.

BinDebloat removes the redundant code by using the second method. It replaces each instruction of the redundant code with an INT 3 instruction which will trigger a software interrupt intended to be used by debuggers. If the program is not running under a debugger, an INT 3 instruction will cause the program to terminate.

V. Evaluation

A. Experimental Setup

We conduct all our evaluations experiments on a workstation equipped with a 32-core 2.2GHz AMD Ryzen Threadripper processor and 128 GB memory. The workstation runs 64-bit Ubuntu 20.04. Our preliminary evaluation is on two popular close-source Windows programs, 7zip file archiver and Adobe Acrobat DC PDF reader.

B. Configuration Dependent Code

Table II shows the number of configuration options, the configuration-dependent instructions, and the total instructions of these programs. As we can see, the two programs have approximately 10% configuration-dependent instructions, which can potentially be removed to reduce the programs' attack surface.

TABLE II: Configuration-Dependent Code Statistics.

Program	# Options	# Config-Dep Instrs	# Total Instrs
7zip	55	11847 (9.8%)	120,889
Acrobat DC	80	63,084 (10.9%)	577,612

C. Attack Surface Reduction

Similar to prior work [15], [16], [22], [27], we use the number of removed ROP gadgets as the metric for the attack surface reduction for a program. We use ROPGadgets to count the number of ROP gadgets of the original programs and that of the programs debloated by BinDebloat.

TABLE III: Attack Surface Reduction.

Program	# Gadgets	% Reduction
7zip	33,410	5.3
Acrobat DC	126,385	8.4

We list the result on attack surface reduction in Table III. Column "# Insts." shows the total number of instructions of a program. Column "# Gadgets" shows the total number ROP gadgets in the program. Column "% Reduction" shows the percentage of the ROP gadgets removed by BinDebloat.

VI. Conclusion

This paper presents BinDebloat, a technique for debloating redundant code from closed-source programs. It focuses on redundant code caused by program configurations. By statically finding code blocks that are control-dependent on configuration options, it identifies a program's redundant code for a set of configuration option values. By removing the redundant code blocks from the program, it reduces the attack surface of the program. The results of our experiment evaluation on real-world Windows programs illustrate that BinDebloat can effectively debloat closed-source programs.

References

[1] Y. Hu, G. Huang, and P. Huang, "Automated reasoning and detection of specious configuration in large systems with symbolic execution," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 719–734.

[2] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 307–319.

[3] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution." in *NDSS*, 2009.

[4] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 497–512.

[5] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[6] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2381–2398.

[7] Z. Huang and G. Tan, "Rapid Vulnerability Mitigation with Security Workarounds," in *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research*, ser. BAR '19, February 2019.

[8] Z. Huang and X. Yu, "Integer overflow detection with delayed runtime test," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES '21. ACM, 2021, pp. 28:1–28:6.

[9] Z. Huang, T. Jaeger, and G. Tan, "Fine-grained program partitioning for security," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21. New York, NY, USA: ACM, 2021, pp. 21–26.

[10] A. Aumpansub and Z. Huang, "Detecting software vulnerabilities using neural networks," in *ICMLC 2021: 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*. ACM, 2021, pp. 166–171. [Online]. Available: https://doi.org/10.1145/3457682.3457707

[11] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1506–1518. [Online]. Available: https://doi.org/10.1145/3510003.3510222

[12] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[13] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1868–1882.

[14] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "Toss: Tailoring online server systems through binary feature customization," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018, pp. 1–7.

[15] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 869–886.

[16] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 1–16.

[17] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 491–502.

[18] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.

[19] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2016, pp. 122–131.

[20] A. A. Ahmad, A. R. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, F. Zaffar, and J. H. Siddiqui, "Trimmer: An automated system for configuration-based software debloating," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3485–3505, 2021.

[21] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2c: Fine-grained configuration-driven system call filtering," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1243–1257.

[22] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.

[23] M. D. Brown and S. Pande, "Carve: Practical security-focused software debloating using simple feature set mappings," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019, pp. 1–7.

[24] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[25] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[26] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, "Slimium: debloating the chromium browser with feature subsetting," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 461–476.

[27] C. Qian, H. Hu, M. Alharthi, S. P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating." in *USENIX Security Symposium*, 2019, pp. 1733–1750.

[28] M. D. Brown and S. Pande, "Is less really more? towards better metrics for measuring security improvements realized through software debloating." in *CSET@ USENIX Security Symposium*, 2019.