REDUCING SOFTWARE'S ATTACK SURFACE WITH CODE DEBLOATING

A Thesis Proposal
Presented to
The Academic Faculty

By

Chenxiong Qian

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

May 2021

REDUCING SOFTWARE'S ATTACK SURFACE WITH CODE DEBLOATING

Thesis committee:

Dr. Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. William R. Harris, Co-Advisor
Principal Scientist
Galois, Inc

Dr. Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Brendan Saltaformaggio
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date approved: April 22, 2020

People in their right minds never take pride in their talents.

Harper Lee

To Kill A Mockingbird

To my wife,

Junqi Zhao,

and my parents,

for all their unconditional love, support and belief in me.

ACKNOWLEDGMENTS

First, I want to express my sincere appreciation to my advisor, Dr. Wenke Lee, who offered me the chance to start my academic career in Georgia Tech. Along my PhD journey, Wenke gave me ample freedom to explore ideas and patiently guided me to pursue the good research direction. I will never forget the words: "Do good work and good results will come." Besides the research mentoring, his self-respect, integrity, and generosity are driving me to be a good and decent man. I hope one day, I can be a good advisor to my future students, as Wenke has been to me.

I am extremely grateful to my co-advisor Dr. William R. Harris for his fully support during my PhD's beginning years and my job hunting. His solid knowledge of program analysis and formal verification has been essentially helpful to my research work. His good taste of research inspired me to cultivate my own research taste. I will never forget the time when he stayed up late helping me on submissions and shared his favorite pizza topping with anchovies.

Another person who had led me to where I am today is Dr. Xiapu Luo. He started advising me when I was an undergraduate and his passion on doing research motivated me to pursue PhD after graduation. I would like to express special thanks to him, as my mentor and friend.

This thesis cannot be done without the help of many brilliant and friendly colleagues. I would like to take the opportunity to thank the following collaborators: Prof. Taesoo Kim, Prof. Hyungjoon Koo, Prof. Hong Hu, Dr. Pak Ho Chung, Dr. Yanick Fratantonio, Prof. Wei Meng, ChangSeok Oh, Mansour Alharthi, Carter Yagemann, Prof. Chengyu Song, Ren Ding, and Prof. Kangjie Lu.

I would like to thank Prof. Taesoo Kim, Prof. Alessandro Orso and Prof. Brendan D. Saltaformaggio for taking time to serve on my thesis committee and providing insightful feedback.

I would like to thank the IISP staffs: Elizabeth Ndongi, Trinh Doan and Gloria Griessman for their efforts to provides such an enjoyable research environment.

Last but not least, I want to thank my wife and my parents for their support, patience and unconditional love throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Current software is designed to support a large spectrum of features to meet various users' needs. However, each end-user only requires a small set of the features to perform required tasks, rendering the software bloated. The bloated code not only hinders optimal execution, but also leads to a larger attack surface. Code debloating technique, which aims to remove the unneeded features' code, has been proposed to reduce the bloated software's attack surface. However, there is a fundamental gap between the features needed by a user and the implementation, so it is challenging to identify the code that only supports the needed features.

Previous works ask end-users to provide a set of sample inputs to demonstrate how they will use the software and generate a debloated version of the software that only contains the code triggered from running the sample inputs. Unfortunately, software debloated by this approach only supports running given inputs, presenting an unusable notion of debloating: if the debloated software only needs to support a fixed set of inputs, the debloating process is as simple as synthesizing a map from the input to the observed output. We call this *Over-debloating Problem*. This dissertation focuses on removing software's unneeded code while providing high robustness for debloated software to run more inputs sharing the same functionalities with the given inputs, with approaches either using heuristics, feature-code map, or code partitioning.

First, the thesis presents RAZOR, which first collects executed code for running the software on the given inputs and then uses heuristics to infer non-executed code related to the given inputs. In the end, RAZOR rewrites the software to keep not only the executed code but also the inferred code, which makes the debloated software support running other inputs besides the given ones. However, in RAZOR, the heuristics are syntax-based and can only infer a limited set of related code, which fails on debloating large-scale and complex software such as web browsers.

Later, the thesis presents SLIMIUM, which uses a feature-code map to debloat the web browser Chromium at feature-level. In SLIMIUM, the feature-code map is initially created from manual analysis and then it is expanded using static program analysis. However, relying on manual efforts to identify features and relevant code is time-consuming and difficult to be applied to other software.

Finally, the thesis presents DEPART, which provides a general approach to debloat large-scale and complex software written with object-oriented programming (OOP) languages without any manual efforts. DEPART performs pure static analysis to automatically partitions a program into distinct groups implementing different features, which is later used for debloating. The key idea of DEPART is to relate the software's code and types (i.e., defined objects sharing unique behaviors in OOP) by analyzing the code's various operations. Based on the relations, we propose several rules to describe the conditions that should be satisfied for including types and code into a same group.

CHAPTER 1
INTRODUCTION

As commodity software is designed to support more features and platforms to meet various users' needs, its size tends to increase in an uncontrolled manner [1, 2]. However, each end-user usually just requires a small subset of these features, rendering the software *bloated*. The bloated code not only leads to a waste of memory, but also opens unnecessary attack vectors. Indeed, many serious vulnerabilities are rooted in the features that most users never use [3, 4]. Therefore, security researchers are beginning to explore software *debloating*, which aims to remove code of unneeded features, as an emerging solution to this problem. However, it is challenging to identify the unneeded code because of the fundamental gap between the unneeded features and the code implementation.

To fill the gap, end-users are asked to provide a set of sample inputs to demonstrate how they will use the software, as in CHISEL [5]. Unfortunately, programs debloated by this approach only supports given inputs, presenting an unusable notion of debloating: if the debloated software only needs to support a fixed set of inputs, the debloating process is as simple as synthesizing a map from the input to the observed output.

In order to practically debloat programs based on user-supplied inputs, we must identify the code that is necessary to completely support required functionalities but is not executed when processing the sample inputs, called *related-code*. Unfortunately, *related-code* identification is difficult. In particular, it is challenging for end-users (even developers) to provide an input corpus that exercises all necessary code that implements a feature. This dissertation presents two approaches to solve this problem: (1) use heuristics to infer code similar to the executed code and keep both executed code and inferred code in the debloated software; (2) create a feature-code map statically and use it as a reference to identify needed features from executed code.

First, I present Razor, in which we design four heuristics that infer *related-code* based on the assumption that code paths with more significant divergence represent less related functionalities. Specifically, given one executed path p, we aim to find a different path q such that 1) q has no different instructions, or 2) q does not invoke new functions, or 3) q does not require extra library functions, or 4) q does not rely on library functions with different functionalities. Then, we believe q has functionalities similar to p and treat all code in q as *related-code*. From 1) to 4), the heuristic includes more and more code in the debloated binary. For a given program, we will gradually increase the heuristic level until the generated program is stable. In fact, our evaluation shows that even the most aggressive heuristic introduces only a minor increase of the final code size. After identifying *related-code* with the heuristics, we develop a binary-rewriting platform to remove unnecessary code and generate a debloated program. To understand the efficacy of RAZOR, we evaluated it on three sets of benchmarks: all SPEC CPU2006 benchmarks, 10 coreutils programs used in previous work, and two real-world large programs, the web browser FIREFOX and the closed-sourced PDF parser FOXITREADER. In our evaluation, we performed tracing and debloating based on one set of training inputs and tested the debloated program using a separate set of functionally similar inputs. Our results show that RAZOR can effectively reduce 70-80% of the original code. At the same time, it introduces only 1.7% overhead to the new binary. We compared RAZOR with CHISEL on debloating 10 coreutils programs and found that CHISEL achieves a slightly better debloating result (smaller code size), but it fails several programs on given test cases. Further, CHISEL introduces exploitable vulnerabilities to the debloated program, such as buffer overflows resulting from the removed boundary checks. RAZOR does not introduce any security issues. We also analyzed the *related-code* identified by our path finder and found that different heuristics effectively improve the program robustness.

While RAZOR uses heuristics to infer *related-code* and generates more stable debloated versions of software than previous systems, it also suffers two major limitations. First, the

heuristics are syntax-based, so they cause both false negatives and false positives. For example, even the most aggressive heuristic, which groups library calls to infer similar functionalities, cannot guarantee that all the *related-code* is identified. Moreover, syntax-based heuristics possibly identify *related-code* that shares different functionalities because developers can implement different features using same library calls. Second, software with stateful modules (e.g., networking communication, local caching) executes different code paths for same inputs (i.e., *non-deterministic code*). Therefore, simply running given inputs to get executed code misses finding *non-deterministic code*. To address these two limitations, I present my second work, SLIMIUM, which aims to debloat one of the most large-scale and complex software Chromium that dominates the web browser market share (i.e., 70%). In this work, we first manually explore the source code and documents to create an initial feature-code map. After that, we do static analysis to seek more relevant code for the features defined in the feature-code map. To do that, for the unmapped code, we calculate a two-dimensional relation vector, $\vec{R} = (r_c, r_s)$, which consists of the following two vector components: i) call invocations (r_c) and ii) similarity between object file names using the hamming distance [6] (r_s). The relation vector serves as a metric on how intensively any two objects are germane to each other. The intuition behind this is that i) it is reasonable to include an object as part of a feature if function calls would be frequently invoked each other, ii) relevant code is likely to be implemented under a similar path name. We tackle the *non-deterministic code* problem by reloading a webpage multiple times until reaching a point when no more new exercised functions are observed with a fixed sliding window (i.e., the length of revisiting that does not introduce a new exercised function). Our experimental results demonstrate the practicality and feasibility of SLIMIUM for 40 popular websites, as on average it removes 94 CVEs (61.4%) by cutting down 23.85 MB code (53.1%) from defined features (21.7% of the whole) in Chromium.

While SLIMIUM can debloat Chromium, but it cannot be easily extended to debloat other large-scale software because it requires having professional knowledge of the source

code and taking much manual efforts to create the initial feature-code map (i.e., 40 working hours in our experiment). Finally, I present DEPART that performs pure static analysis to automatically partition a program into groups implementing distinct features. Based on the observation that large-scale and complex software is usually developed with object-oriented programming (OOP) languages, in which types are defined to represent objects sharing unique behavior. First, DEPART extracts the type reference graphs in the program. Second, DEPART conduct static analysis on the code to identify the various operations (e.g., object allocation, attribute access, member function implementation, etc.) performed among code and types. We address the challenge of building data dependence relations among code by proposing a novel static analysis with a combination of points-to analysis and taint analysis. With the static analysis results, DEPART builds the relation graphs to assigns relations among code and types. In the end, DEPART defines several rules to describe the conditions that should be satisfied to include code and types into a group implementing the same feature. We have evaluated DEPART on debloating Chromium and our experimental results show that DEPART identifies more features than the manual analysis used in SLIMIUM and DEPART removes more code than SLIMIUM when debloating for 40 popular websites.

The dissertation is organized as follows:

- I first illustrate the motivation for removing code of unneeded features in software and the limitations exist in previous debloating systems in Chapter §2.
- I then introduce the related work for identifying/removing unused features' code and partitioning code in Chapter §3.
- In Chapter §4, I present our first work that debloats bloated binaries using heuristics to infer code that is not executed but share similar functionalities with given inputs.
- In Chapter §5, I present our second work that debloats the web browser Chromium using a prebuilt feature-code map.

- In Chapter §6, I present a novel approach for partitioning a program automatically into groups implementing distinct features and apply it on debloating Chromium.
- In Chapter §7, I discuss the limitations of the proposed work and present the future work.
- Finally, I conclude this dissertation in Chapter §8.

CHAPTER 2

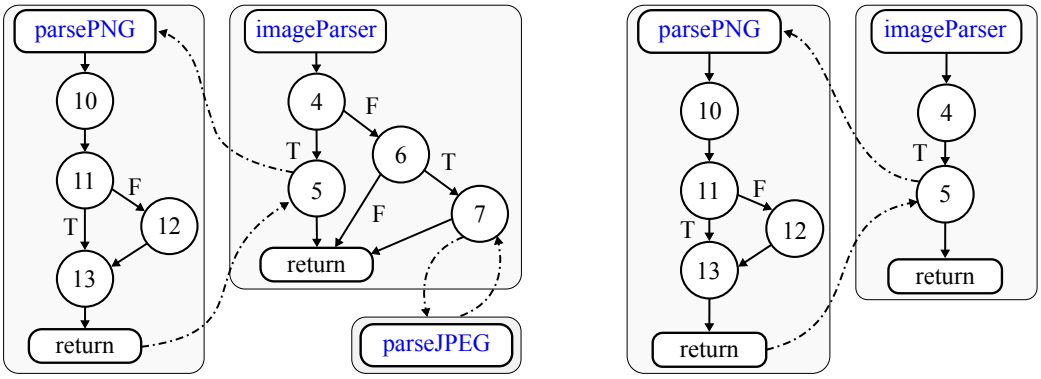
MOTIVATION

Current software is designed to support more features and platforms to meet various users' needs, its size tends to increase in an uncontrolled manner [1, 2]. However, each end-user usually just requires a small subset of these features, rendering the software *bloated*. For an example, there are around 80% features in Chromium are only used by less than 10% of the websites in the wild [7]. The bloated code not only leads to a waste of memory, but also opens up unnecessary attack vectors. Indeed, many serious vulnerabilities are rooted in the features that most users never use [3, 4]. Therefore, security researchers are beginning to explore software *debloating* as an emerging solution to this problem. I will show a motivation example that contains bloated code to describe how previous debloating systems debloat the program and their limitations.

```
1  #define MAX_SIZE 0xffff
2  #define ALIGN(v,a) (((v+a-1)/a)*a)
3  void imageParser(char *options, char *file_name) {
4    if (!strcmp(options, "PNG"))
5      parsePNG(file_name);
6    else if (!strcmp(options, "JPEG"))
7      parseJPEG(file_name);
8  }
9  void parsePNG(char *file_name) {
10   char * img = (char *)malloc(MAX_SIZE + 16);
11   if ((img % 16) != 0)
12     img = ALIGN(img, 16);
13   readToMem(img, file_name);
14 }
15 void parseJPEG(char *file_name) { ... }
```

Figure 2.1: A bloated image parser.

Figure 2.1 shows a bloated program, which is designed to parse image files in different formats. Based on the user-provided `options` (line 4 and 6), the program invokes function



(a) Original control-flow graph. (b) Debloated control-flow graph

Figure 2.2: Debloating an image parser. (a) shows the code of the bloated image parser, where the program invokes different functions to handle `PNG` or `JPEG` files based on the options. The control-flow graphs before and after debloating are shown in (b) and (c).

`parsePNG` to parse PNG images (line 5) or invokes function `parseJPEG` to handle JPEG images (line 7). In function `parsePNG`, the code first allocates memory to hold the image content and saves the memory address in `img` (line 10). Then it makes sure `img` is aligned to 16-bytes with the macro `ALIGN` (line 11 and 12). Finally, it invokes function `readToMem` to load the image content from file into memory for further processing. Function `parseJPEG` has a structure similar to `parsePNG`, so we skip its details.

Although the program in Figure 2.1 merely supports two image formats, it is still bloated if the user only uses it to process `PNG` files. For example, screenshots on iPhone devices are always in PNG format [8]. In this case, the code is bloated with the unnecessary `JPEG` parser, which may contain security bugs [9]. Attackers can force it to process malformed `JPEG` images to trigger the bug and launch remote code execution. In real-world software ecosystem, we can easily find document readers (*e.g.*, Preview on MacOS) that support obsolete formats (*e.g.*, PCX, Sun Raster, TGA). We can debloat these programs to reduce their code sizes and attack surfaces.

Previous systems assume the set of given test cases is complete and only keeps the code triggered from running the test cases. However, it is impossible to provide test cases that cover all related-code of the required functionalities. In this case, some related-code will not be triggered. If we simply remove all never-executed code, the program functionality

will be broken. For example, the code at lines 11 and 12 of Figure 2.1 will make sure the pointer `img` is aligned to 16. Based on the concrete execution context, the return value of `malloc` (at line 10) may or may not satisfy the alignment requirement. If the execution just passes the check at line 11, the simple method will delete line 12 for the minimal code size. However, if the later execution expects an aligned `img`, the program will show unexpected behavior or even crash. We call this *Over-debloating Problem*.

To solve this problem, I first propose RAZOR that uses four heuristics to infer *related-code* and debloats software on instruction-level, then I propose SLIMIUM that debloats large-scale software on feature-level using a feature-code map built from manual analysis and static analysis. In the end, I propose an approach to automatically partition a program by doing summarization based a program's self-defined data structures' operations. With the summarization result, we can automatically build a program's feature-code map without any manual efforts.

CHAPTER 3
RELATED WORK

It is crucial to identify accurately unneeded code to avoid *Over-debloating Problem*. To that end, prior works largely fall into three categories to identify unused features: a) binary analysis (*e.g.*, static or dynamic analysis), b) supplementary information with the help of a compilation toolchain, and c) machine learning techniques.

Debloating with binary analysis One of the early works based on static analysis is Code-Freeze [10]. It presents a technique, dubbed "code stripping and image freezing" that eliminates imported functions not in use at load time, followed by freezing the remaining code to protect it from further corruption (*e.g.*, injecting code). Because it targets executable binaries whose sources are unavailable, this approach performs code removal atop a conservative static analysis. DamGate [11] introduces a framework to customize features at runtime. It leverages a handful of existing tools to build a call graph through both static and dynamic analyses. In a similar vein, TRIMMER [12] begins with identifying unnecessary features based on an user-defined configuration, followed by eliminating corresponding code from interprocedural analysis statically.

Meanwhile, Shredder [13] aims to filter out potentially dangerous arguments of well-known APIs (*e.g.*, assembly functions). It first collects the range of API parameters that a benign application takes and then enforces a policy to obey the allowable scope of the parameters from initial analysis. For example, a program would be suspended upon a call invocation with unseen arguments. Both FACE-CHANGE [14] and Kernel tailoring [15, 16] apply the concept of debloating to the kernel. The former makes each application view its own shrinking version of the kernel, facilitating dynamic switching at runtime across different process contexts, whereas the latter automatically generates a specific ker-

nel configuration used for compiling the tailored kernel. Meanwhile, both the bloat-aware design [17] and JRed [18] apply program customization techniques to Java-based applications; whereas Babak et al. [19] propose a debloating technique for web applications.

Debloating with supplementary information Another direction toward code debloating takes advantage of a compilation toolchain to obtain additional information. PieceWise [20] introduces a specialized compiler that assists in emitting call dependencies and function boundaries within the final binary as supplementary information. The modified loader then takes two phases (i.e., page level and function level) to invalidate unneeded code at load time.

Debloating with machine learning techniques Recent advancements in debloating leverage various machine learning techniques to identify unused code or features. CHISEL [21] produces a transformed version that removes unneeded features with reinforcement learning. Because it relies on test cases as an input to explore internal states, it might suffer from incorrect results when running a mutation that encounters an unexpected state. Hecate [22] leverages deep learning techniques to identify features and corresponding functions. It uses both a recursive neural network (RNN) to compute semantic representation (*e.g.*, unique embedding vector per each opcode and operand) and a convolutional neural network (CNN) for a function mapping as a multi-class classifier. Binary control flow trimming [23] introduces a contextual control flow graph (CCFG) that enables the expression of explicitly user-unwanted features among implicitly developer-intended ones, learning a CCFG policy based on runtime traces. BlankIt [24] applies machine learning to predicate functions needed at load time.

Other efforts to reduce attack surface The work of Snyder et al. [25] leverages Web API standards to limit the functionality of a website. However, it has two major differences: a) the attack surface only contains standard Web APIs without considering non-web

features, and b) the hardening mechanism lies in disabling specific features by intercepting JavaScript, implemented as one of the browser extensions. The actual implementation code still resides in memory; thus it could be circumvented [26] with an expected access.

Anh et al. [27] propose bloat metrics for the first time to systematically quantify the bloatness of each program, library, and system library. CARVE [28] takes an approach of debloating unused source code, which requires both open source and a rebuilding process. Cimplifier [29] demonstrates that a container image could shrink its size up to 95%, preserving its original functionalities. Recently, Microsoft released `ApplicationInspector` [30], an attack surface analysis tool based on known patterns, automatically identifying third-party software components that might impact security. Other efforts include code removal based on configurable features for applications [31], system call specialization [32], and its policy generation [33] for containers.

CHAPTER 4

HEURISTIC-BASED APPROACH

To address *Over-debloating Problem*, I first propose RAZOR that uses heuristics to identify *related-code* effectively and debloats post-deployment software without source code. I will first demonstrate the overview and design of RAZOR in Section §4.1 and then show the evaluation in Section §4.2.

4.1 Overview and Design

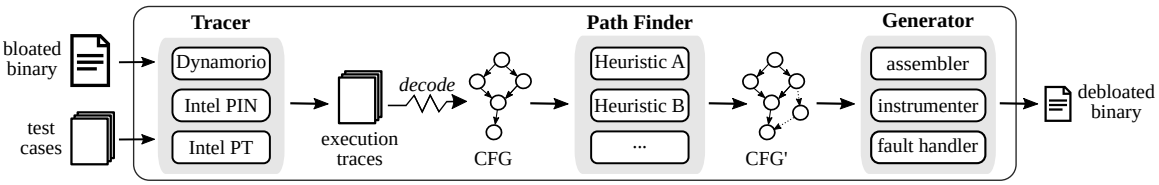


Figure 4.1: Overview of RAZOR. It takes in the bloated program binary and a set of test cases and produces a minimal, functional binary. TRACER collects the program execution traces with given test cases and converts them into a control-flow graph (CFG). PATHFINDER utilizes control-flow-based heuristics to expand the CFG to include more *related-code*. Based on the new CFG, GENERATOR generates the debloated binary.

Figure 4.1 shows an overview of our post-deployment debloating system, RAZOR. Given a bloated binary and a set of test cases that trigger required functionalities, RAZOR removes unnecessary code and generates a debloated binary that supports all required features with minimal code size. To achieve this goal, RAZOR first runs the binary with the given test cases and uses TRACER to collect execution traces (§4.1.1). Then, it decodes the traces to construct the program's CFG, which contains only the executed instructions. In order to support more inputs of the same functionalities, PATHFINDER expands the CFG based on our control-flow heuristics (§4.1.2). The expanded CFG contains non-executed instructions that are necessary for completing the required functionalities. In the end, with

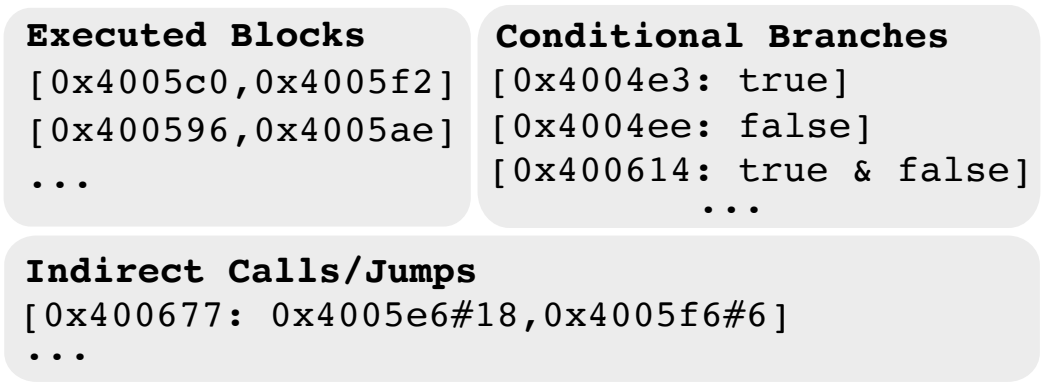


Figure 4.2: A snippet of the collected trace. It includes the range of each executed basic block, the taken/non-taken of each condition branch, and the concrete target of indirect jumps/calls. We also record the frequency of each indirect jump/call target (after #).

the expanded CFG, GENERATOR rewrites the original binary to produce a minimal version that only supports required functionalities (§4.1.3).

4.1.1 Execution Trace Collection

TRACER executes the bloated program with given test cases and records the control-flow information in three categories: (1) executed instructions, including their memory addresses and raw bytes; (2) the taken or non-taken of conditional branches, like `je` that jumps if equal; (3) concrete targets of indirect jumps and calls, like `jmpq *%rax` that jumps to the address indicated by register `%rax`. Our TRACER records the raw bytes of executed instructions to handle dynamically generated/modified code. However, instruction-level recording is inefficient and meanwhile most real-world programs only contain static code. Therefore, TRACER starts with basic block-level recording that only logs the address of each executed basic block. During the execution, it detects any dynamic code behavior, like both writable and executable memory region (*e.g.*, just-in-time compilation [34]), or overlapped basic blocks (*e.g.*, legitimate code reuse [35]), and switches to the instruction-level recording to avoid missing instructions. A conditional branch may get executed multiple times and finally covers one or both targets (*i.e.*, the fall-through target and the jump target). For indirect jump/call instructions, we log all executed targets and count their frequencies.

Figure 4.2 shows a piece of collected trace. It contains two executed basic blocks, one at address `0x4005c0` and another at `0x400596`. The trace also contains three conditional branch instructions: the one at `0x4004e3` only takes the `true` target; the one at `0x4004ee` only takes the `false` target; the one at `0x400614` takes both targets. One indirect call instruction at `0x400677` jumps to target `0x4005e6` for 18 times and jumps to target `0x4005f6` for six times. As the program only has static code, TRACER does not include the instruction raw bytes.

We find that it is worthwhile to use multiple tools to collect the execution trace. First, no mechanism can record the trace completely and efficiently. Software-based instrumentation can faithfully log all information but introduces significant overhead [36, 37, 38]. Hardware-based logging can record efficiently [39] but requires particular hardware and may not guarantee the completeness (*e.g.*, data loss in INTEL PT [40]). Second, program executions under different tracing environments will show divergent paths. For example, DYNAMORIO always expands the file name to its absolute path, leading to different executed code in some programs (*e.g.*, `vim`). Therefore, we provide three different implementations with different software and hardware mechanisms. End-users can choose the best one for their requirement or even merge traces from multiple tools for better code coverage.

CFG construction. With the collected execution traces, RAZOR disassembles the bloated binary and constructs the partial control-flow graph (CFG) in a reliable way. Different from previous works that identify function boundaries with heuristics [41, 42, 43, 44, 45], RAZOR obtains the accurate information of instruction address and function boundary from the execution trace. For example, we can find some of all possible targets of indirect jumps and calls.

Starting from such reliable information, we are able to identify more code instructions [46]. For conditional branch instructions, both targets are known to us. Even if one target is not executed, we can still reliably disassemble it. For indirect jumps, we can identify potential jump tables with specific code patterns [47]. For example, `jmpq *0x4e65a0(,%rdx,8)` indicates a jump table starting from address `0x4e65a0`. By identifying more instructions,

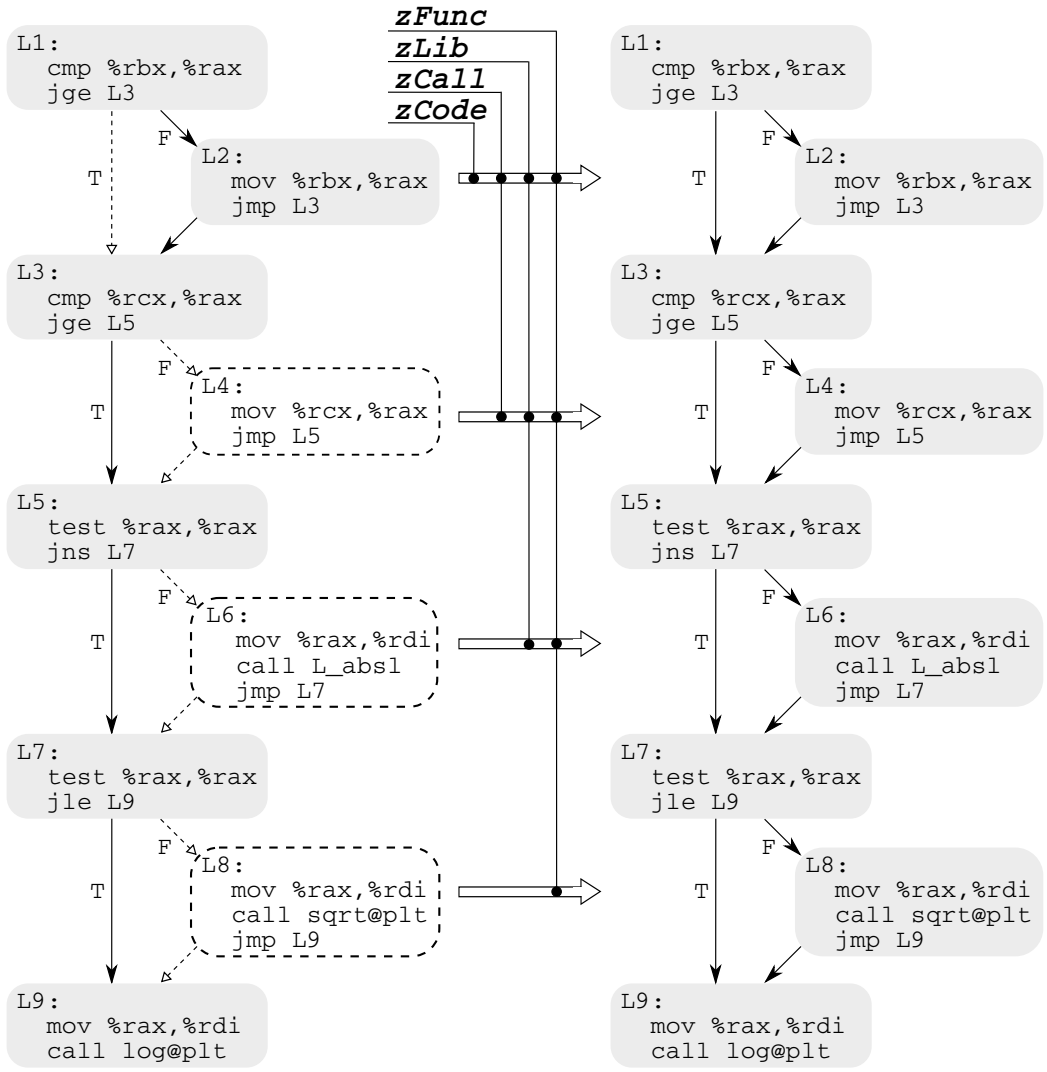


Figure 4.3: Identifying *related-code* with different heuristics. Dashed branches and blocks are not executed and thus are excluded from the left CFG, while others are executed.

we are able to include them in the binary if our heuristic treats them as *related-code*.

4.1.2 Heuristic-based Path Inference

Considering the challenge of generating test cases to cover all code, we believe no perfect method can completely identify all missed *related-code*. As the first work trying to mitigate the problem, we adopt the *best-effort heuristic* approach to include more *related-code*. Next, we present these heuristics one by one, from the conservative one (including less code) to the aggressive one (including more code):

(1) Zero-code heuristic (zCode). This heuristic adds new edges (*i.e.*, jumps between basic blocks) into the CFG. For conditional branch instructions that only have one target taken (the fall-through target or the jump target), PATHFINDER checks whether the non-taken target is already in the CFG (*i.e.*, reached through other blocks). If so, PATHFINDER permits the jump from this instruction to the non-taken target. This heuristic does not add any new instructions and thus will not affect the code reduction.

Figure 4.3 shows an example of *related-code* identification with heuristics, with the original CFG on the left and the expanded CFG on the right. The code is designed to calculate `log(sqrt(absl(max(rax,rbx,rcx))))`. Dashed branches and blocks are not executed during tracing, while others are executed. The original execution path is `L1`→`L2`→`L3`→`L5`→`L7`→`L9`. Blocks `L4`, `L6`, `L8`, and the branch `L1`→`L3` are missed in the original CFG. With the zCode heuristic, PATHFINDER adds branch `L1`→`L3` into the new CFG, as `L3` is the non-taken branch of the conditional jump `jge L3` in `L1` and it is already reached from `L2` in the current CFG.

(2) Zero-call heuristic (zCall). This heuristic includes alternative execution paths that do not trigger any function call. With this heuristic, PATHFINDER starts from the non-taken target of some conditional branches and follows the control-flow information to find new paths that finally merge with the executed ones. If such a new path does not include any `call` instructions, PATHFINDER includes all its instructions to the CFG. When PATHFINDER walks through non-executed instructions, we do not have the accurate information for stable disassembling or CFG construction. Instead, we rely on existing mechanisms [47, 43] to perform binary analysis. When applying the zCall heuristic on the example in Figure 4.3, PATHFINDER further includes block `L4`, and path `L3`→`L4`→`L5`, as this new path merges with the original one at `L5` and does not contain any `call` instruction.

(3) Zero-libcall heuristic (zLib). This heuristic is similar to zCall, except that PATHFINDER includes the alternative paths more aggressively. The new path may have `call` instructions that invoke functions within the same binary or external functions that have been executed.

Algorithm 1: Path-finding algorithm.

```
   Input: CFG - the input CFG; libcall_groups - the library call groups.
   Output: CFG′ - the expanded CFG
   CFG′ ← CFG
   /* iterate over each conditional branch                                    */
1  for cnd_br ∈ CFG:
2      nbb = get_non_taken_branch(cnd_br)
3      if nbb == NULL: continue
4      if heuristic ≥ zCode and nbb ∈ CFG:
5          CFG′ = CFG′ ∪ {cnd_br→nbb}
6      paths = get_alternative_paths(CFG′, nbb)
7      for p ∈ paths:
8          include = false
9          if   heuristic == zCall:  include = !has_call(p)
10         elif heuristic == zLib:   include = !has_new_libcall(p)
11         elif heuristic == zFunc:
12             include = !has_new_func(CFG′, p, libcall_groups)
13         if include:
14             CFG′ = CFG′ ∪ p
```

However, zLib does not allow calls to non-executed external functions. In Figure 4.3, with this heuristic, PATHFINDER adds block `L6` and path `L5`→`L6`→`L7` to the CFG, as that path does not have any call to non-executed external functions.

(4) Zero-functionality heuristic (zFunc). This heuristic further allows including non-executed external functions as long as they do not trigger new high-level functionalities. To correlate library functions with functionalities, we check their descriptions and group them manually. For `libc` functions, we classify the ones that fall into the same subsection in [48] to the same group. For example, `log` and `sqrt` are in the subsection `Exponentiation and Logarithms`, and thus we believe they have similar functionalities. With this heuristic, PATHFINDER includes block `L8` and path `L7`→`L8`→`L9`, as `sqrt` has a functionality similar to the executed function `log`.

Algorithm 1 shows the steps that PATHFINDER uses to find *related-code* that completes functionalities. For each conditional branch in the input CFG (line 1), the algorithm invokes the function `get_non_taken_branch` to get the non-taken branch (line 2). If both branches have been taken, the algorithm proceeds to the next conditional branch (line 3). Otherwise, PATHFINDER starts to add code depending on the given heuristic (line 4 to

14). If the non-taken branch is reachable in the current CFG (line 4), zCode enables the new branch in the output CFG (line 5). If the heuristic is more aggressive than zCode, PATHFINDER first gets all alternative paths that start from the non-taken branch and finally merges with some executed code (line 6). Then, it iterates over all paths (line 7) and calls corresponding checking functions (*i.e.*, `has_call`, `has_new_libcall`, and `has_new_func`) to check whether or not the path should be included (line 9 to 12). In the end, PATHFINDER adds the path to the output CFG if it satisfies the condition (line 14).

4.1.3 Debloated Binary Synthesization

With the original bloated binary and the expanded CFG, GENERATOR synthesizes the debloated binary that exclusively supports required functionalities. First, it disassembles the original binary following the expanded CFG and generates a pseudo-assembly file that contains all necessary instructions. Second, GENERATOR modifies the pseudo-assembly to create a valid assembly file. These modifications symbolize basic blocks, concretize indirect calls/jumps, and insert fault handling code. Third, it compiles the assembly file into an object file that contains machine code of the necessary instructions. Fourth, GENERATOR copies the machine code from the object file into a new code section of the original binary. Fifth, GENERATOR modifies the new code section to fix all references to the original code and data. Finally, GENERATOR sets the original code section non-executable to reduce the code size. We leave the original code section inside the debloated program to support the potential read from it (*e.g.*, jump tables in code section for implementing `switch` [49]).

Basic Block Symbolization

We assign a unique label to each basic block and replace all its references with the label. Specifically, we create the label `L_addr` for the basic block at address `addr`. Then, we scan all direct jump and call instructions and replace their concrete target addresses with corresponding labels. In this way, the assembler will generate correct machine code regardless

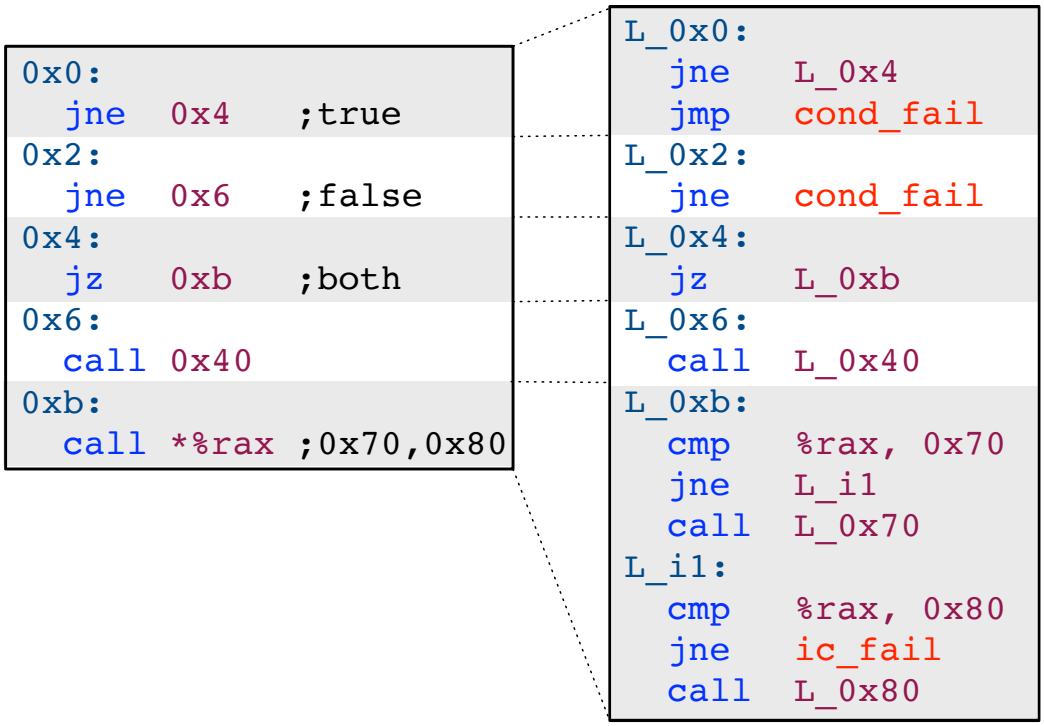


Figure 4.4: Synthesize debloated assembly file. Each basic block is assigned a unique label; indirect calls are expanded with comparisons and direct calls; fault handling code is inserted.

of how we manipulate the assembly file. Figure 4.4 shows an assembly file before and after the update, illustrating the effect of basic block symbolization. Before the update, all call and jump instructions use absolute addresses, like `jne 0x6` in basic block `0x0`. After the symbolization, the basic block at `0x6` is assigned the label `L_0x6`, while instruction `jne 0x6` is replaced with `jne L_0x6`. Similarly, instruction `call 0x40` in block `0x06` is replaced with `call L_0x40`. One special case is the conditional branch `jne 0x6` in basic block `0x2`. In the extended CFG, it only takes the fall-through branch, which means that jumping to block `0x6` should not be allowed in the debloated binary. Therefore, instead of replacing `0x6` with symbol `L_0x6`, we redirect the execution to the fault handling code `cond_fail` (will discuss in §4.1.3). Note that basic block symbolization only updates explicit use of basic block addresses, *i.e.*, as direct call/jump targets. We handle the implicit address use, like saving function address into memory for indirect call, with the indirect call/jump concretization.

Indirect Call/Jump Concretization

Indirect call/jump instructions use implicit targets that are loaded from memory or calculated at runtime. We have to make sure all possible targets point to the new code section. For the sake of simplicity, we use the term `indirect call` to cover both indirect calls and indirect jumps.

With the execution traces, GENERATOR is able to handle indirect calls in two ways. The first method is to locate constants from the original binary that are used as code addresses and replace them with the corresponding new addresses, as in [41, 42]. However, this method requires a heavy tracing process that records all execution context and a time-consuming data-flow analysis. Therefore, it is impractical for large programs. The second method is to perform the address translation before each indirect call, as in [47]. In particular, we create a map from the original code addresses to the new ones. Before each indirect call, we map the old code address to the new one and transfer the control-flow to the new address.

Our GENERATOR takes a method similar to the second one, but with different translations for targets within the same module (named local targets) and targets outside the module (named global targets). For local targets, we define a concrete policy for each indirect call instruction. Specifically, we replace the original call with a set of compare-and-call instructions, one for each local target that is executed by *this* instruction at tracing. Then, we call the new address of the matched old addresses. Global targets have different addresses in multiple runs because of the address space layout randomization (ASLR). We use a per-module translation table to solve this problem. Different from previous work that creates a translation table for all potential targets in the module [47], our translation table contains only targets that are ever invoked by other modules. At runtime, if the target address is outside the current module, we use a global translation function to find the correct module and look up its translation table to get the correct new address to invoke.

Figure 4.4 gives an example of indirect call concretization. In the execution trace, in-

struction `call *%rax` in block `0xb` transfers control to function at `0x70` and `0x80`. Our concretization inserts two `cmp` instructions, one to compare with the address `0x70` and another to compare with `0x80`. For any successful comparison, GENERATOR inserts a direct call to transfer the control-flow to the corresponding new address.

Security benefit. Our design achieves a stronger security benefit on control-flow protection over previous methods. For example, the previous work binCFI [47] uses a map to contain all valid code addresses, regardless of which instruction calls them. Thus, any indirect call instruction can reach all possible targets, making the protection vulnerable to existing bypasses [50, 51, 52]. Our design is functionally equivalent to creating one map for each indirect call, which contains both the targets obtained from the trace and the targets inferred by our PATHFINDER. For inter-module indirect calls, we limit the targets to a small set that is ever invoked by external modules. In this way, attackers who try to change the control flow will have fewer choices, and the debloated binary will be immune to even advanced attacks.

Frequency-based optimization. Depending on the number of executed targets, we may insert many compare-and-call instructions that will slow the program execution. For example, one indirect call instruction in `perlbench` benchmark of SPEC CPU2006 has at least 132 targets, and each target is invoked millions of times. To reduce the overhead, we rank all targets with their execution frequencies and compare the address with high-frequent targets first. The targets inferred from heuristics have a frequency of zero. With this optimization, we can reduce the overhead significantly.

Fault Handling

Running a debloated binary may reach removed code or disabled branches for various reasons, such as a user's temporal requirement for extra functionalities or malicious attempts to run unnecessary code. We redirect any such attempt to a fault handler that exits the execution and dumps the call stack. Specifically, for conditional jump instructions with

only one target taken, we intercept the branch to the non-taken target to hook any attempt of the invalid jump. Similarly, for indirect call instructions, if no allowed target matches the runtime target, we redirect the execution to the fault handler.

Figure 4.4 includes examples of hooking failed conditional jumps and indirect calls. For instruction `jne 0x4` in block `0x0`, we insert `jmp cond_fail` to redirect the branch to the fall-through target to the fault handler `cond_fail`. Similarly, we update instruction `jne 0x6` with `jne cond_fail` to prevent jumping to the non-executed target. For conditional branch `jz 0xb` which has both targets taken, we do not insert any code. For instruction `call *%rax`, we insert code `jne ic_fail` in the case that all allowed targets are different from the real-time one.

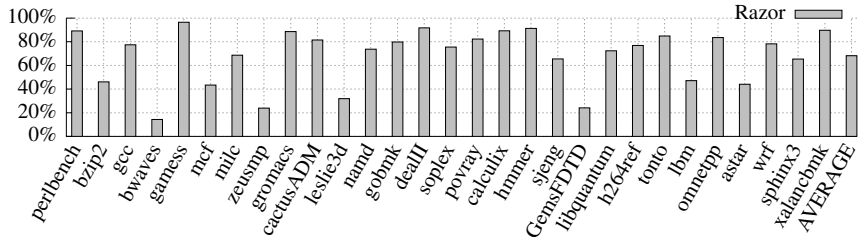
4.2 Evaluation

In this section, we perform extensive evaluation in order to understand RAZOR regarding the following aspects:

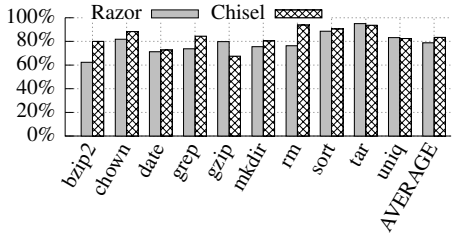
- **Code reduction.** How much code can RAZOR reduce from the original bloated binary (§4.2.1)
- **Functionality.** Does the debloated binary support the functionalities in given test cases (§4.2.2) How effective is PATHFINDER in finding complementary code? (§4.2.3)
- **Security.** Does RAZOR reduce the attack surface of the debloated binaries? (§4.2.4)
- **Performance.** How much overhead does RAZOR introduce into the debloated binary? (§4.2.5)
- **Practicality.** Does RAZOR work on commonly used software in the real world? (§4.2.6)

Experiment setup. We set up three sets of benchmarks to evaluate RAZOR: 29 SPEC CPU2006 benchmarks, including 12 C programs, seven C++ programs, and 10 Fortran programs; 10 coreutils programs used in the CHISEL paper[1] [5]; the web browser FIRE-

[1] We appreciate the help of CHISEL authors for sharing the source code and their benchmarks.



(a) SPEC CPU2006



(b) CHISEL benchmarks

Figure 4.5: Code size reduction on two benchmarks. We use RAZOR to debloat both SPEC CPU2006 benchmarks and CHISEL benchmarks without any path finding and achieve 68.19% and 78.8% code reduction. CHISEL removes 83.4% code from CHISEL benchmarks.

FOX and the close-source PDF reader FOXITREADER. We use the software-based tracing tools that rely on DYNAMORIO and PIN to collect the execution traces of SPEC and CHISEL benchmarks, to get accurate results; for the complicated programs FIREFOX and FOXITREADER, we use the hardware-based tracing tool (relying on INTEL PT) to guarantee the execution speed to avoid abnormal behaviors. We ran all the experiments on a 64-bit Ubuntu 16.04 system equipped with Intel Core i7-6700K CPU (with eight 4.0GHz cores) and 32 GB RAM.

4.2.1 Code Reduction

We applied RAZOR on SPEC CPU2006 benchmarks and CHISEL benchmarks to measure the code size reduction. For SPEC benchmarks, we treated the `train` dataset as the user-given test cases. For CHISEL benchmarks we obtained test cases from the paper's authors. We did not apply any heuristics of path finding for this evaluation. As RAZOR works on binaries, we cannot measure the reduction of source code lines. Instead, we compare

the size of the executable memory region before and after the debloating, specifically, the program segments with the executable permission. Figure 4.5a shows the code reduction of SPEC benchmarks debloated by RAZOR. Figure 4.5b shows the code reduction of CHISEL benchmarks, debloated by CHISEL and RAZOR.

On average, RAZOR achieves 68.19% code reduction for SPEC benchmarks and 78.8% code reduction for CHISEL benchmarks. Especially for `dealII`, `hmmer`, `gamess`, and `tar`, RAZOR removes more than 90% of the original code. For `bwaves`, `zeusmp`, and `GemsFDTD`, RAZOR achieves less than 30% code reduction. We investigated these exceptions and found that these programs are relatively small and the `train` datasets already trigger most of the code.

Meanwhile, CHISEL achieves 83.4% code reduction on CHISEL benchmarks. For seven programs, CHISEL reduces more code than RAZOR, while RAZOR achieves higher code reduction than CHISEL for the other three programs. CHISEL tends to remove more code as long as the execution result remains the same. For example, variable initialization code always gets executed at the function beginning. CHISEL will remove it if the variable is not used in the execution, while RAZOR will keep it in the debloated binary. Although CHISEL performs slightly better than RAZOR on code reduction, we find that the debloated binaries from CHISEL suffer from robustness issues (§4.2.2) and security issues (§4.2.4).

4.2.2 Functionality Validation

We ran the debloated binaries in CHISEL benchmarks against given test cases to understand their robustness. For each benchmark, we compiled the original source code to get the original binary and compiled the debloated source code from CHISEL to get the CHISEL binary. Then, we used RAZOR to debloat the original binary with given test cases, generating the RAZOR binary. Next, we ran the original binary, the CHISEL binary, and the RAZOR binary again with the test cases. We examine the execution results to see whether the required functionalities are retained in the debloated binaries.

Table 4.1: Failed test cases by RAZOR binaries and CHISEL binaries. CHISEL failed some tests with different reasons: **W**rong operations, **I**nfinite loop, **C**rashes, and **M**issing output. For RAZOR binaries, we show the heuristic that makes the program pass all tests.

Program	Version	# of Tests	Failed by Chisel				Failed by Razor
			W	I	C	M	
bzip2	1.0.5	6	**2**	–	**2**	–	– (zLib)
chown	8.2	14	–	–	–	–	– (zFunc)
date	8.21	50	**5**	–	**3**	–	– (zLib)
grep	2.19	26	–	–	–	**6**	– (zLib)
gzip	1.2.4	5	–	**1**	–	–	– (zLib)
mkdir	5.2.1	13	–	–	–	**1**	– (zLib)
rm	8.4	4	**2**	–	–	–	– (zFunc)
sort	8.16	112	–	–	–	–	– (zCall)
tar	1.14	26	**3**	–	–	**4**	– (zCall)
uniq	8.16	16	–	–	–	–	– (zCall)

Table 4.1 shows the validation result. RAZOR binaries produce the same results as those from the original binaries for all test cases of all programs (the last column), showing the robustness of the debloated binaries. Surprisingly, CHISEL binaries only pass the tests of three programs (*i.e.*, `chown`, `sort`, and `uniq`) and trigger some unexpected behaviors for the other seven programs. Considering that CHISEL verifies the functionality of the debloating binary, such a low passing rate is confusing. We checked these failed cases and the verification process of CHISEL and found four common issues.

Wrong operation. The debloated program performs unexpected operations. For examples, `bzip2` should decompress the given file when the test case specifies the `-d` option. However, the binary debloated by CHISEL always decompresses the file regardless of what option is used. We suspect that CHISEL only uses one test case of decompression to debloat the program and thus removes the code that parses command line options.

Infinite loop. CHISEL may remove loop condition checks, leading to infinite loops. For example, `gzip` fails one test case because it falls into a loop in which CHISEL drops the condition check. We believe the reason is that the test case used by CHISEL only iterates the loop one time. The verification step of CHISEL should identify this problem. However, we found that the verification script adopts a small timeout (*e.g.*, 0.1s) and treats any timeout

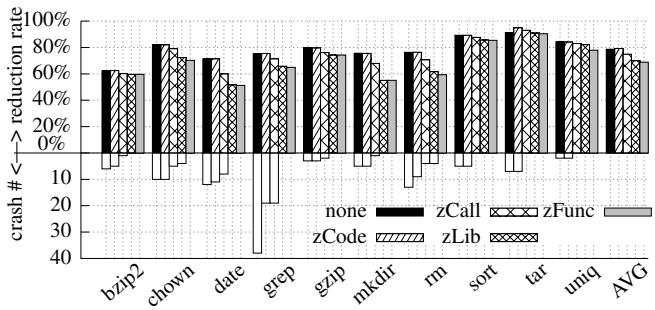


Figure 4.6: Path finding on CHISEL benchmarks with different heuristics. The top part is the code reduction, while the bottom part is the number of crashes. 'none' means no heuristic is used.

as a successful verification. Therefore, it cannot detect any infinite loops.

Crashes. The debloated binary crashes during execution. For example, `date` crashes three test cases because CHISEL removes the check on whether the parameters of `strcmp` are `NULL`. `bzip2` crashes three test cases for the same reason.

Missed output. CHISEL removes code for printing out on stdout and stderr, leading to missed results. For example, `grep` fails six test cases, as the binary does not print out any result even through it successfully finds matched strings. We find that in the verification script of CHISEL, all output of the debloated binaries is redirected to the `/dev/null` device. Therefore, it cannot detect any missing or inconsistent output.

4.2.3 Effectiveness of Path Finding

We use two sets of experiments to evaluate the effectiveness of PATHFINDER on finding the *related-code* of required functionalities. First, we use RAZOR to debloat programs with different heuristics, from the empty heuristic to the most aggressive zFunc heuristic, aiming to find the least aggressive heuristic for each program. Second, we perform N-fold cross validation to understand the robustness of our heuristic. In this subsection, we focus on the first experiment and leave the N-fold cross validation in §4.2.6.

We tested RAZOR on CHISEL benchmarks as follows: (1) design training inputs and testing inputs that cover the same set of functionalities; (2) trace programs with the train-

ing inputs and debloat them with none, zCode, zCall, zLib, and zFunc heuristics; (3) run debloated binaries on testing inputs and record the failed cases. The setting of evaluating PATHFINDER is given in Table A.1 of Appendix A. We use the same options for training inputs and testing inputs to make sure that the debloated binaries are tested for the same functionalities as those triggered by the training inputs. The difference is the concrete value for each option or the file to process. For example, when creating folders with `mkdir`, we use various parameters of the option `-m` for different file mode sets. For program `bzip2` and `gzip`, we use different files for training and testing.

Figure 4.6 presents our evaluation result, including the code reduction (the top half) and the number of failed test cases (the bottom half) under different heuristics. We can see that debloating with a more aggressive heuristic leads to more successful executions. All binaries generated without any heuristic fail on some testing inputs. `grep` fails on all 38 testing inputs, while `chown` and `rm` fail more than half of all tests. The zCode heuristic helps mitigate the crash problem, like making `grep` work on 19 test cases. However, all generated binaries still fail some inputs. The zCall heuristic further improves the debloating quality. For program `sort`, `tar`, and `uniq`, it avoids all previous crashes. With the zLib heuristic, only two programs (*i.e.*, `chown` and `rm`) still have a small number of failures. In the end, debloating with the zFunc heuristic reduces all crashes in all programs.

Interestingly, although aggressive heuristics introduce more code to the debloated binary (shown in the top of Figure 4.6), they do not significantly decrease the code reduction. Without any heuristic, the average code reduction rate of 10 programs is 78.7%. The number is reduced by $-0,4\%$, 3.8%, 8.8%, and 12.6% when applying zCode, zCall, zLib, and zFunc heuristics, respectively. Therefore, even with the most aggressive zFunc heuristic, the code reduction does not decrease heavily. At the same time, all crashes are resolved, showing the benefits of applying heuristics. Note that the zCode heuristic slightly increases the code reduction over the no heuristic case, as it enables more branches of conditional jumps, which in turn reduces the instrumentation of failed branches.

```
int fillbuf(...) { ...
  if (minsize <= maxsize_off)
    if (...) ...
  newalloc = newsize+ ...;
}
```

Figure 4.7: A crash case reduced by applying zCode heuristic.

```
int fts_safe_changedir(..,){
  if (dir) {
    tmp=strcmp(dir,".."); ...
  } ...
}
```

Figure 4.8: A crash case reduced by applying zFunc heuristic.

```
int compare(line *a,line *b) {
  alen = a->length - 1UL;
  blen = b->length - 1UL;
  if (alen == 0UL) {
    diff = -(blen != 0UL);
  } else {
    if (blen == 0UL) {
      diff = 1;
    } else { ... }
}}
```

Figure 4.9: A crash case reduced by applying zCall heuristic.

```
int main(...) { ...
  fail = make_dir(..);
  if (!fail) {
    if (!create_parents) {
      if (!dir_created) {
        tmp_7=gettext("error");
        error(0,17,tmp_7,tmp_6);
        fail = 1;
        ...
}}}}
```

Figure 4.10: A crash case reduced by applying zLib heuristic.

We investigated the failed cases mitigated by different heuristics and show some case studies as follows:

(1) The **zCode** heuristic enables the non-taken branch for executed conditional jumps. Figure 4.7 shows part of the function `fillbuf` of program `grep` that fails if we do not use the zCode heuristic. The training inputs always trigger the `true` branch of the condition at line 2 and jump to line 3, which in turn reach line 4. However, in the execution of testing inputs, the conditional at line 2 takes the false branch (i.e., `minsize > maxsize_off`) and triggers the jump from line 2 to line 4. This branch is not allowed from execution traces. The zCode heuristic enables this branch, as line 4 has been reached in the previous execution.

(2) The **zCall** heuristic includes alternative paths that do not trigger any `call` instructions. Figure 4.9 shows an example where the zCall heuristic helps include necessary code in the debloated binary. Function `compare` in program `sort` uses a sequence of comparisons to find whether two text lines are different. Since the training inputs have no empty lines, the

condition at line 4 and line 7 always fails. However, the testing inputs contain empty lines, which makes these two conditional jumps take the `true` branches. The zCode heuristic adds lines 5 and 8 and related branches to the debloated program, which effectively avoids this crash.

(3) The **zLib** heuristic allows extra calls to native functions or library functions if they have been used in traces. It helps avoid a crash in program `mkdir` when we use the debloated binary to change the file mode of an existing directory. Figure 4.10 shows the related code, which crashes because of the missing code from line 6 to line 9. Since `mkdir` does not allow changing the file mode of an existing directory, the code first invokes function `gettext` to get the error message and then calls library function `error` to report the error. The zLib heuristic includes this path in the binary because both `gettext` and `error` are invoked by some training inputs.

(4) The **zFunc** heuristic includes alternative paths that invoke similar library functions. Figure 4.8 shows the code that causes `rm` to fail without this heuristic. When `rm` deletes a folder that contains both files and folders, it triggers the code at line 3 to check whether it is traversing to the parent directory. Since the training inputs never call `strcmp`, the debloated binary fails even with the zLib heuristic. However, the training inputs ever invoke function `strncmp`, which has the functionality similar to `strcmp` (*i.e.*, string comparison). Therefore, the zFunc heuristic adds this code in the debloated binary.

The results show that PATHFINDER effectively identifies *related-code* that completes the functionalities triggered by training inputs. It enhances the robustness of the debloated binaries while retaining the effectiveness of code reduction.

4.2.4 Security Benefits

We count the number of reduced bugs to evaluate the security benefit of our debloating. For each program in the CHISEL benchmark, we collected all its historical vulnerabilities, including the ones shown in the current version and the ones only in earlier versions. For

Table 4.2: Vulnerabilities before and after debloating by RAZOR and CHISEL. ✓ means the binary is vulnerable to the CVE, while ✗ mean it is not vulnerable. CVEs with ∗ are evaluated in [5].

Program	CVE	Orig	Chisel	Razor
bzip2-1.0.5	CVE-2010-0405	✓		
	CVE-2011-4089*	✗		
	CVE-2008-1372	✗	✔	
	CVE-2005-1260	✗	✔	
chown-8.2	CVE-2017-18018*	✓	✘	✘
date-8.21	CVE-2014-9471*	✓	✘	
grep-2.19	CVE-2015-1345*	✓	✘	✘
	CVE-2012-5667	✗	✔	
gzip-1.2.4	CVE-2005-1228*	✓	✘	✘
	CVE-2009-2624	✓		
	CVE-2010-0001	✓	✘	✘
mkdir-5.2.1	CVE-2005-1039*	✓		
rm-8.4	CVE-2015-1865*	✓		
sort-8.16	CVE-2013-0221*	✗		
tar-1.14	CVE-2016-6321*	✓	✘	
uniq-8.16	CVE-2013-0222*	✗		

the former bugs, we check whether the buggy code has been removed by the debloating process. If so, the debloating process helps avoid related attacks. For the latter bugs, we figure out whether their patches are retained in the debloated binary. If not, the debloated process makes the program vulnerable again. Table 4.2 shows our evaluation result, including 16 CVEs related to CHISEL benchmarks. 13 bugs are shown in the current version, and 10 of them are evaluated in [5] (followed by *). Three bugs only exist in older versions (*i.e.*, CVE-2010-0405, CVE-2009-2624, and CVE-2010-0001).

RAZOR successfully removes four CVEs from the original binaries and does not introduce any new bugs. Specifically, CVE-2017-18018 in `chown`, CVE-2015-1345 in `grep`, CVE-2005-1228 and CVE-2010-0001 in `gzip` are removed in the debloated binaries. Six vulnerabilities from `bzip`, `date`, `gzip`, `mkdir`, `rm`, and `tar` remain, as the test cases execute related vulnerable code. Another six vulnerabilities are not caused by the binary itself. For example, CVE-2011-4089 is caused by the race condition of the bash script `bzexe`, not by the `bzip2` binary. Therefore, RAZOR will not disable such bugs.

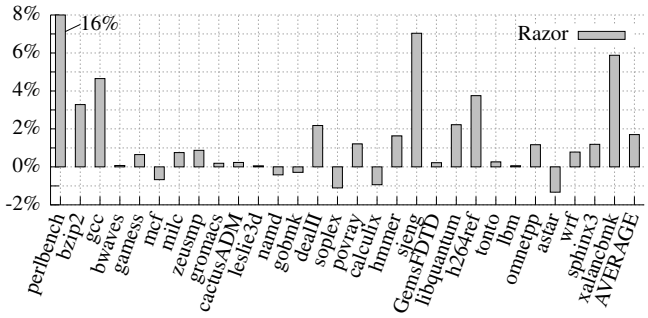


Figure 4.11: Performance overhead by RAZOR on SPEC CPU®2006 benchmarks. The average overhead is 1.7%.

With a more aggressive code removal policy, CHISEL disables two more CVEs than RAZOR, but unfortunately brings three old bugs to the debloated binaries. Specifically, CHISEL removes the vulnerable code of CVE-2014-9471 from `date` and the code of CVE-2016-6321 from `tar`. Meanwhile, it removes the patches of CVE-2008-1372 and CVE-2005-1260 in `bzip2`, and CVE-2012-5667 in `grep`, rendering the debloated binaries vulnerable to these already-fixed bugs.

Compared to CHISEL, RAZOR removes the bloated code in a conservative way. Although such strategy may hinder removing more bugs, but it also helps avoid new bugs in the debloated binary. This result is consistent with our findings in §4.2.2, where CHISEL achieves higher code reduction but fails some expected functionalities.

4.2.5 Performance Overhead

Efficient Debloating. On average, RAZOR takes 1.78 seconds to debloat CHISEL benchmarks, 8.51 seconds for debloating FIREFOX, and 50.42 seconds to debloat FOXITREADER As a comparison, CHISEL has to spend one to 11 hours to debloat the relatively small CHISEL benchmarks. Therefore, RAZOR is a practical debloating tool.

Runtime Overhead. We measured the performance overhead introduced by RAZOR to SPEC benchmarks and show the result in Figure 4.11. On average, RAZOR introduces 1.70% overhead to debloated programs, indicating its efficiency for real-world deployment. The highest overhead occurs on the debloated `perlbench` binary, which slows the execu-

tion by 16%. We inspected the debloated programs and confirmed that the indirect call concretization is the main source of the performance overhead. With the indirect call concretization, one indirect call instruction is replaced by several comparison and direct calls. For `perlbench`, some indirect call instructions have more than 100 targets. Correspondingly, RAZOR introduces a large number of `if-else` there, leading to a high performance overhead. We deployed the frequency-based optimization and reduced the overhead from over 100% to the current 16%. We plan to use binary search to replace current one-by-one matching in order to further reduce the overhead.

4.2.6 Debloating Real-world Programs

To evaluate the practicality, we used RAZOR to debloat two widely used software programs – the web browser FIREFOX and the closed-sourced PDF reader FOXITREADER. For FIREFOX, we ran RAZOR to load the top 50 Alexa websites [53]. We randomly picked 25 websites as the training inputs and used the other 25 websites as the testing inputs. For FOXITREADER, we ran RAZOR to open and scroll 55 different PDF files that contain tables, figures, and JavaScript code. We randomly picked 15 of them as the training inputs and used the other 40 files as the testing inputs.

Code reduction and functionality. Table 4.3 shows the code reduction rate and the number of failed cases of debloated binaries with different path-finding heuristics. Both FIREFOX and FOXITREADER require at least the zLib heuristic to obtain crash-free binaries, with 60.1% and 87.0% code reduction, respectively. Without heuristics, FIREFOX fails on 13 out of 25 websites and FOXITREADER fails on 39 out of 40 PDF files. The zCode heuristic helps reduce FOXITREADER crashes to 10 PDF files and increases the code reduction by avoiding fault-handling instrumentation. The zLib and the zFunc heuristic eliminate all crashes. Compared with the non-heuristic debloating, the zLib heuristic only decreases the code reduction rate by 7.5% for FIREFOX and by 2.8% for FOXITREADER. Therefore, it is worth using this heuristic to generate robust binaries.

Table 4.3: Debloating FIREFOX and FOXITREADER with RAZOR, together with diffferent path-finding heuristics.

Heuristic	FireFox		FoxitReader	
	crash-sites	reduction	crash-PDFs	reduction
none	13	67.6%	39	89.8%
zCode	13	68.0%	10	89.9%
zCall	2	63.1%	5	89.4%
zLib	0	60.1%	0	87.0%
zFunc	0	60.0%	0	87.0%

Table 4.4: Per-site browser debloating

Type	Site	Reduction	Heuristic	Benefits
Banking	bankofamerica.com	69.4%	zCall	+6.3%
	chase.com	69.6%	zCall	+6.5%
	wellsfargo.com	68.8%	zCall	+5.7%
	all-3	68.1%	zCall	+5.0%
E-commerce	amazon.com	71.4%	none	+3.8%
	ebay.com	70.7%	none	+3.1%
	ikea.com	70.6%	none	+3.0%
	all-3	70.4%	none	+2.8%
Social Media	facebook.com	70.8%	zCall	+7.7%
	instagram.com	71.6%	zCall	+8.5%
	twitter.com	74.0%	none	+6.4%
	all-3	71.8%	none	+4.2%

Performance overhead. We ran the debloated FIREFOX (with zLib) on several benchmarks and found that RAZOR introduces -2.1%, 1.6%, 0%, and 2.1% overhead to Octane [54], SunSpider [55], Dromaeo-JS [56], and Dromaeo-DOM [57] benchmarks. For FOXITREADER, we did not find any standard benchmark to test the performance. Instead, we used the debloated binaries to open and scroll the testing PDF files and did not find any noticeable slowdown.

Application – per-site browser isolation. As one application of browser debloating, we can create minimal versions that support particular websites, effectively achieving per-site isolation [58, 59, 60] . For example, the bank can provide its clients a minimal browser that only supports functionalities required by its website while exposing the least attack surface. To measure the benefit of the per-site browser, we applied RAZOR on three sets of popular

Table 4.5: N-fold validation of zLib heuristic on FIREFOX. First, we randomly split Alexa's Top 50 websites into five groups, and select two groups (20 websites) as the training set and others (30 websites) as the test set for 10 times. Second, we randomly split the 50 website into 10 groups, and select five groups (25 websites) as the training set, and others (25 websites) as the test set for 10 times.

Train/Test	ID	#Failed	Reduction	failed websites
20/30	T10	1	59.3%	wordpress.com
	T11	0	59.3%	
	T12	1	59.3%	wordpress.com
	T13	1	59.3%	twitch.tv
	T14	1	59.3%	wordpress.com
	T15	1	59.5%	wordpress.com
	T16	2	59.5%	twitch.tv, wordpress.com
	T17	1	59.3%	twitch.tv
	T18	1	59.3%	twitch.tv
	T19	2	59.6%	wordpress.com, twitch.tv
25/25	T00	0	59.3%	
	T01	2	59.1%	wordpress.com, twitch.tv
	T02	2	59.3%	wordpress.com, twitch.tv
	T03	2	59.1%	wordpress.com, twitch.tv
	T04	0	59.2%	
	T05	1	59.1%	aliexpress.com
	T06	0	59.2%	
	T07	0	59.1%	
	T08	2	59.3%	wordpress.com, twitch.tv
	T09	0	59.1%	

and security-sensitive websites: banking websites, websites for electronic commerce, and social media websites. Table 4.4 shows the debloating result, the used path-finding heuristic and the security benefits over the general debloating in Table 4.3. As we can see, the banking websites can benefit with at least 5.0% code reduction for the per-site minimal browser. The E-commerce websites will have around 3.0% extra code reduction, a little less because of its high requirement on user interactions. Surprisingly, social media websites can benefit by up to 8.5% extra code reduction and at least 4.2% when supporting all three websites. We believe the minimal web browser through binary debloating is a practical solution for improving web security.

N-fold Cross Validation of Heuristics

To further evaluate the effectiveness of our heuristics, we conducted N-fold cross validation on FIREFOX with the zLib heuristic, as it is the least aggressive heuristic that renders FIREFOX crash-free.We performed two sets of evaluations and show the result in Table 4.5. First, we randomly split Alexa's Top 50 websites into five groups, 10 websites per group. We picked two groups (20 websites) for training and used the remaining 30 websites for testing. We performed this evaluation 10 times. The result in the table shows that during one test with ID T11, the debloated FIREFOX successfully loads and renders 30 testing websites. The debloated FIREFOX fails two websites (6.7%) seven times and fails one website (3.3%) two times. Second, we randomly split Alexa's Top 50 websites into 10 groups, five websites per group. We randomly picked five groups (25 websites) for training and used the others (25 websites) for testing. We performed this evaluation 10 times. The result shows that, in five times, the debloated FIREFOX loads and successfully renders the tested 25 websites. The debloated FIREFOX fails one (4%) website one time and fails two websites (8%) four times. The code size reduction is consistently round 60%. These results show that our heuristics are effective for inferring non-executed code with similar functionalities of training inputs. Among all the tests, only three websites trigger additional code and the program gracefully exits with warning information. We plan to check these websites to understand the failure reasons.

We also manually checked what code of FIREFOX was removed. We find that code related to features such as record/replay, integer/string conversion, compression/decompression are removed.

CHAPTER 5

FEATURE-CODE MAP APPROACH

While the heuristic-based approach shown in §4 suffers from both false positives and false negatives, so it does not work for large-scale software with dynamic inputs. In this Chapter, I propose SLIMIUM that uses a feature-code map to debloat one of the large-scale and complex software, Chromium. I will introduce the overview of SLIMIUM in Section §5.1 and the design in detail in Section §5.2.

5.1 Overview

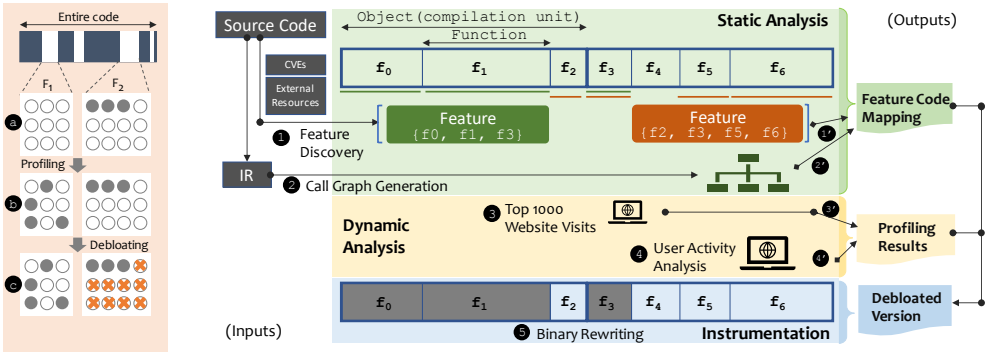


Figure 5.1: High-level overview of SLIMIUM. It leverages a concept of feature subsetting (feature as a unit of debloating) to guide a binary instrumentation as feedback on top of feature-code mapping and profiling results.

Figure Figure 5.1 shows an overview of SLIMIUM for debloating Chromium. SLIMIUM consists of three main phases: i) feature-code mapping generation, ii) prompt website profiling based on page visits, and iii) binary instrumentation based on i) and ii).

Feature-Code Mapping To build a set of unit features for debloating, we investigate source code [61], previously-assigned CVEs pertaining to Chromium, and external resources [62, 63] for the Web specification standards (Step ❶ in Figure Figure 4.1). Ta-

ble Table 5.1 summarizes 164 features with four different categories. Once the features have been prepared, we generate a *feature-code map* that aids further debloating from the two sources (❶' and ❷'). From the light-green box in Figure Figure 5.1, consider the binary that contains two CUs to which three and four consecutive binary functions (i.e., $\{f_0 - f_2\}$ and $\{f_3 - f_6\}$) belong, respectively. The initial mapping between a feature and source code relies on a manual discovery process that may miss some binary functions (i.e., from the source generated at compilation). Then, we apply a new means to explore such missing functions, followed by creating a call graph on the IR (Intermediate Representation) (Step ❷, Section §5.2.2).

Website Profiling The light-yellow box in Figure Figure 5.1 enables us to trace exercised functions when running a Chromium process. SLIMIUM harnesses a website profiling to collect non-deterministic code paths, which helps to avoid accidental code elimination. As a baseline, we perform differential analysis on exercised functions by visiting a set of websites (Top 1000 from Alexa [64]) multiple times (Step ❸). For example, we mark any function non-deterministic if a certain function is not exercised for the first visit but is exercised for the next visit. Then, we gather exercised functions for target websites of our interest with a defined set of user activities (Step ❹). During this process, profiling may identify a small number of exercised functions that belong to an unused feature (i.e., initialization). As a result, we obtain the final profiling results that assist binary instrumentation (❸' and ❹').

Binary Rewriting The final process creates a debloated version of a Chromium binary with a feature subset (Step ❺ in Figure Figure 5.1). In this scenario, the feature in the green box has not been needed based on the feature-code mapping and profiling results, erasing the functions $\{f_0, f_1, f_3\}$ of the feature. As an end user, it is sufficient to take Step ❹ and ❺ for binary instrumentation where pre-computed feature-code mapping and profiling results are given as supplementary information.

Table 5.1: Chromium features as a debloating unit (#: count).

Class	Features (#)	Functions (#)	Function Size (KB)	CVEs (#)	Feature Policy Directives (#)	Experimental Flags (#)
HTML5	6	8,103	1,721	15	0	0
JS API	100	71,082	17,204	57	25	15
Non-web	57	62,594	21,303	77	0	0
Wasm	1	1,189	869	4	0	0
Total	164	142,968	41,097	153	25	15

5.2 Design

5.2.1 Feature Set for Chromium Debloating

We begin with investigating all Web APIs to group them into different features from the approach in Snyder et al. [25], and exploring the Chromium's source code structure to include other features with an absence of Web APIs. Besides, we utilize external resources [63, 62] that list comprehensive features to define our final feature set for debloating. Note that we have excluded i) glue code that is commonly shared among multiple features and ii) code pertaining to fundamental security mechanisms such as SOP (Same Origin Policy) and CSP (Content Security Policy), which means that these security relevant features will be always retained in a debloated version of Chromium regardless of our profiling phase (§5.2.3). In Table Table 5.1, we define 164 Chromium features that can be harnessed as a debloating unit, classifying them into four categories: JS API, HTML5, Non-web, and Wasm. Note that `wasm` (Web assembly) is the only feature that does not belong to HTML5, JS API, or the standard Web specifications. Interested readers can find further details regarding unit features in the Appendix (Table Table B.2). In summary, a few notable statistical values are as follows: i) 153 CVEs reside in 42 debloatble features (25% of all the features), ii) 25 Feature Policy directives are included as part of our feature set, iii) 15 features can be enabled with an experimental flag, eight of which are defined as Feature Policy directives [1].

[1] accelerometer, ambient-light-sensor, fullscreen, magnetometer, gyroscope, vr, publickey-credentials, and xr-spatial-tracking

JavaScript API Chromium offers Web APIs that interact with web contents through JavaScript interfaces. In particular, we utilize `caniuse` [63] to classify the JavaScript APIs because it actively keeps track of browser-and-version-specific features as collective intelligence. Some of them have been combined due to a common implementation (i.e., `Blob` constructing and `Blob` URLs as a `Blob` API), resulting in 100 sub-categories.

HTML5 As the latest HTML version, HTML5 defines a rich feature set including audio, video, vector graphics (i.e., SVG, canvas), MathML, and various form controls as part of HTML by default. We define six major features that cause either a large code base or previous vulnerabilities (i.e., known CVEs). Recently, MarioNet [65] has demonstrated a new class of attacks that sorely relies on HTML5 APIs (i.e., a feature of service workers) in modern browsers, leading successfully unwanted operations.

Non-web Features Our finding shows that there are a few Chromium-browser-specific features such as devTools, extensions, and PDF that have been exposed to various attacks in the past years (Table B.3). To exemplify, we could find 26 CVEs pertaining to a PDF feature alone. Additionally, we define each third-party component as a feature, assuming external code has a minimal dependency on each other. Indeed, this assumption holds for our features because their core implementations are often mutually exclusive. For example, few call invocations have been discovered among each other under the `third_party` directory based on our call graph analysis. Note that we have excluded a few of them when the feature is heavily employed by other parts such as `protobuf`.

5.2.2 Feature-Code Mapping

Generating a feature-code map is a key enabler to make our debloating approach feasible. In this section, we describe how to create such mapping in a reliable and efficient manner. To this end, we introduce a concept of a relation vector to seek more relevant code for a certain feature.

Manual Feature-Code Discovery

To determine the corresponding code to each feature, we begin with a manual investigation on source files that implement a certain feature, which is worthwhile because Chromium often offers well-structured directories and/or file names and test suites. For example, the test set of the `battery` feature resides in `external/wpt/battery-status` and `battery-status` under the directory of `blink/web_tests` that contains a collection of various test suites. With additional exploration, we could infer the implementation for that feature is within `battery` under the directory of `blink/renderer/modules` that contains a collection of various renderer modules.

Feature-Oriented Call Graph Generation

Function Identifier Once the above initial mapping is complete, SLIMIUM constructs a call graph based on IR functions. Recall that we aim to directly remove binary functions on top of the mapping information; hence SLIMIUM instruments the final Chromium binary by assigning a unique identifier for each IR function at its build. The discrepancy between IR and binary functions happens because of i) object files irrelevant to the final binary at compilation (i.e., assertions that ensure a correct compilation process or platform-specific code) and ii) de-duplication at link time [66] (i.e., a single constructor or destructor of a class instance would be selected if redundant) [2]. It is noted that we can safely discriminate all binary functions because they are a subset of IR functions.

Indirect Call Targets As Chromium is mainly written in a C++ language, there are inherently a large number of indirect calls, including both virtual (91.8%) [67] and non-virtual calls. Briefly, we tackle identifying indirect call targets using the following two techniques: i) backward data analysis for virtual calls and ii) type matching for non-virtual calls. In case of failing the target in a backward data analysis for a virtual call invocation, we attempt to

[2] In our experiment, almost half of IR functions were disappeared.

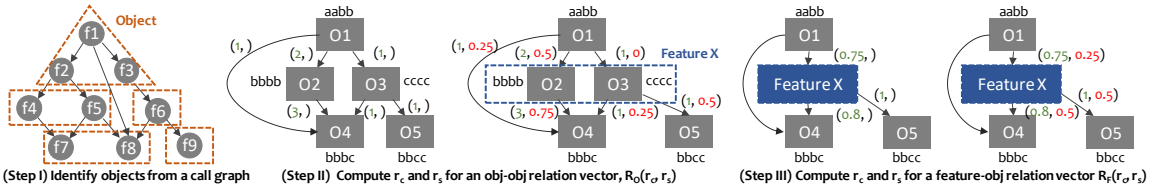


Figure 5.2: Feature-code mapping with relation vectors that enable the inference of relevant object files for a certain feature.

use type matching. Note that it is infeasible to obtain all indirect call targets with full accuracy.

Feature-Code Mapping with Relation Vectors

At this point, we have a large directed call graph (nodes labeled with a function identifier and edges that represent caller-callee relationships) and an initial mapping between a feature and corresponding source files. Even with the mapping information alone, it is possible to learn partial binary functions that belong to relevant object files; however, there are quite a few missing links, including binary functions from sources generated at compilation.

To seek more relevant object files for each feature, we define a two-dimensional *relation vector*, $\vec{R} = (r_c, r_s)$, which represents the following two vector components: i) call invocations (r_c) and ii) similarity between two object file names using the hamming distance [6] (r_s). The relation vector serves as a metric on how intensively any two objects are germane to each other. The intuition behind this is that i) it is reasonable to include an object as part of a feature if function calls would be frequently invoked each other, ii) relevant code is likely to be implemented under a similar path name, and iii) a non-deterministic code path problem (i.e., exceptions) can be minimized by including all functions within an object.

Figure Figure 5.2 illustrates three phases that automatically infer relevant code at the object level. First, as in Step I, we group a set of binary functions that belong to the same object (i.e., `f7` and `f8` with a dotted-line area). In this example, there are five objects grouped with nine functions total. Second, we build another directed graph for object dependencies (Step II) based on the edges from the previous function call graph. Each

edge defines an *object-object relation vector*, $\vec{R_O} = (r_c, r_s)$, between two objects (nodes). For $\vec{R_O}$, each component can be computed as the number of call invocations and a hamming distance value. For instance, the R_O between `01` and `02` can be represented as $(2, 0.5)$ because of two function invocations (i.e., (`f2`) $\rightarrow$ (`f4`, `f5`)) and a hamming distance value of 0.5 from the two object names (i.e., `aabb` and `bbbb`). Third, we consider a feature on top of the object dependency graph (Step III). The initial mapping from manual discovery comes into play, which identifies relevant objects for a certain feature. Suppose the two objects, `02` and `03`, belong to Feature X (dotted-line in blue). Now, we compute another relation vector, a *feature-object relation vector*, $\vec{R_F} = (r_c, r_s)$ for the edges only connected to the feature. For $\vec{R_F}$, the r_c component represents the rate of call invocations between the feature and the surrounding objects (that have edges) whereas the r_s component is an amortized hamming distance value between the object name and the object name(s) that belong to the feature. In this example, the R_F between Feature X and the object `01` would be $(0.75, 0.25)$ because $r_c = \frac{2+1}{1+2+1}$ and $r_s = \frac{0.5+0}{2}$, respectively. Hence, the result can be interpreted as follows: the `01` has a high outgoing call invocation rate to the functions in Feature X; however, its object name is not close enough. Algorithm Algorithm 3 briefly shows pseudo-code on how to explore any relevant objects for further debloating using relation vectors.

Note that we open both r_c and r_s as hyperparameters of our heuristic algorithm to determine the proximity between a feature and an object, ranging from 0 to 1. In our experiment, we use the value of 0.7 for both parameters (See Section §5.3.3 in detail).

5.2.3 Prompt Webpage Profiling

We employ a dynamic profiling technique to complement static analysis (i.e, feature-code mapping) because the granularity of our debloating approach aims at the function level. Various tools are available such as Dynamorio [36] and Intel Pin [68] to obtain exercised functions at runtime through dynamic instrumentation. However, instrumented code in-

evitably introduces considerable performance degradation that leads to a huge lag when running a giant application such as Chromium. Although a hardware-assisted means such as Intel PT [69] significantly addresses the performance issue during a trace, decoding the trace result is non-negligible (i.e., a couple of hours when visiting a certain webpage for a few seconds in Chromium).

Due to the impracticality of prior approaches, we devise a new means to trace with an in-house instrumentation, recording exercised functions akin to an AFL's [70] approach of a global coverage map. First, we allocate a shared memory that can cover the entire IR functions, a superset of binary functions. Second, we build Chromium so that it could mark every exercised function in the shared memory. Third, we *promptly* obtain the list of exercised functions by parsing the whole bits in the shared region after visiting a target webpage. We have not experienced any slowdown with the instrumented version of Chromium during our profiling because our instrumentation requires merely a few bit-operations, a memory read, and a memory write for each function.

As discussed before, it is highly likely to trigger divergent execution paths due to Chromium's inherent complexity even when loading the same page again. We tackle this problem simply by reloading a webpage multiple times until reaching a point when no more new exercised functions are observed with a fixed sliding window (i.e., the length of revisiting that does not introduce a new exercised function; 10 in our case). With the Top 1000 Alexa websites [64], we had to visit a main page of each site approximately 172 times on average. Note that we leave this sliding window open as a hyperparameter.

5.3 Evaluation

In this section, we evaluate SLIMIUM on a 64-bit Ubuntu 16.04 system equipped with Intel(R) Xeon(R) E5-2658 v3 CPU (with 48 2.20 GHz cores) and 128 GB RAM. In particular, we assess SLIMIUM from the following three perspectives:

- Correctness of our discovery approaches: How well does a relation vector technique

discover relevant code for feature-code mapping (Section §5.3.1) and how well does a prompt web profiling unveil non-deterministic paths (Section §5.3.2)?

- Hyperparameter exploration: What would be the best hyperparameters (thresholds) to maximize code reduction while preserving all needed features reliably (Section §5.3.3)?

- Reliability and practicality: Can a debloated variant work well for popular websites in practice (Section §5.3.4)? In particular, we have quantified the amount of code that can be removed (Section §5.3.4) from feature exploration (Section §5.3.4). We then highlight security benefits along with the number of CVEs discarded accordingly (Section §5.3.4).

5.3.1 Code Discovery with a Relation Vector

Our investigation for the initial mapping between features and source code requires approximately 40 hours of manual efforts for a browser expert. We identify 164 features and 6,527 source code files that account for 41.1 MB (37.4%) of the entire Chromium binary. Then, we apply a relation vector technique to seek more objects, as described in §5.2.2. To exemplify, the initial code path for `Wasm` only indicates the directory of `v8/src/wasm`, however, our technique successfully identifies relevant code in `v8/src/compiler`. In this case, although the object names under that directory differ from the beginning directory (i.e., `wasm`), the call invocation component of the vector correctly deduces the relationship.

Figure Figure 5.3 concisely depicts that varying threshold pairs of name similarity (r_s) and call invocation (r_c) are inversely proportional to additional code discovery at large. The dark blue area on the lower left corner holds relatively a high value (i.e., 57.0 MB for $(0.5, 0.5)$) whereas the yellow area on the upper right holds a low one (i.e., 42.3 MB for $(0.9, 0.9)$). Similarly, Figure Figure 5.4 shows a distribution of additional code discovery rate with a handful of different threshold sets from $(0.5, 0.5)$ to $(0.9, 0.9)$. The boxplot

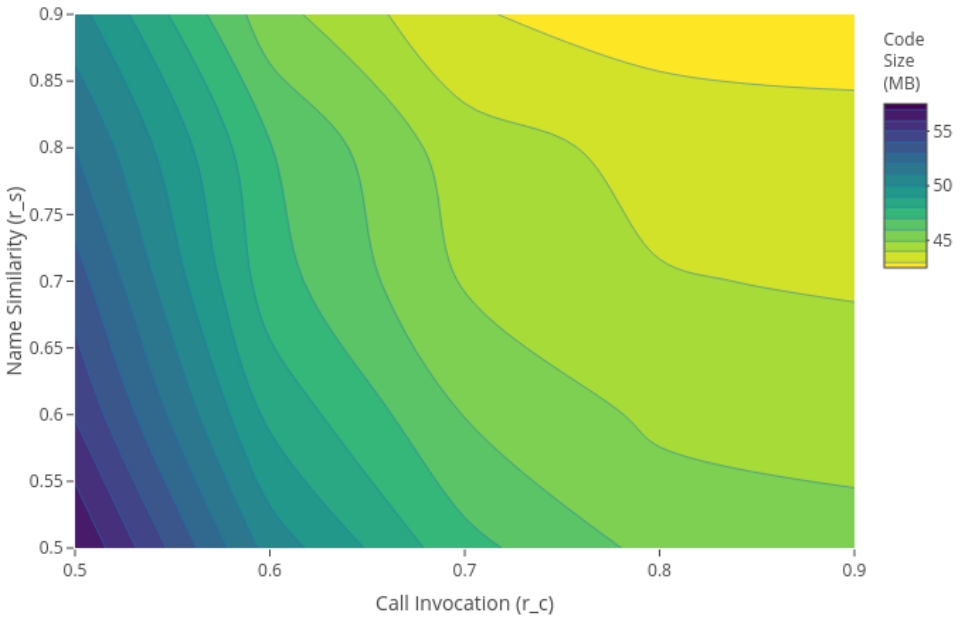


Figure 5.3: Contour plot of additionally discovered code size with a set of different relation vectors $\vec{R} = (r_c, r_s)$.

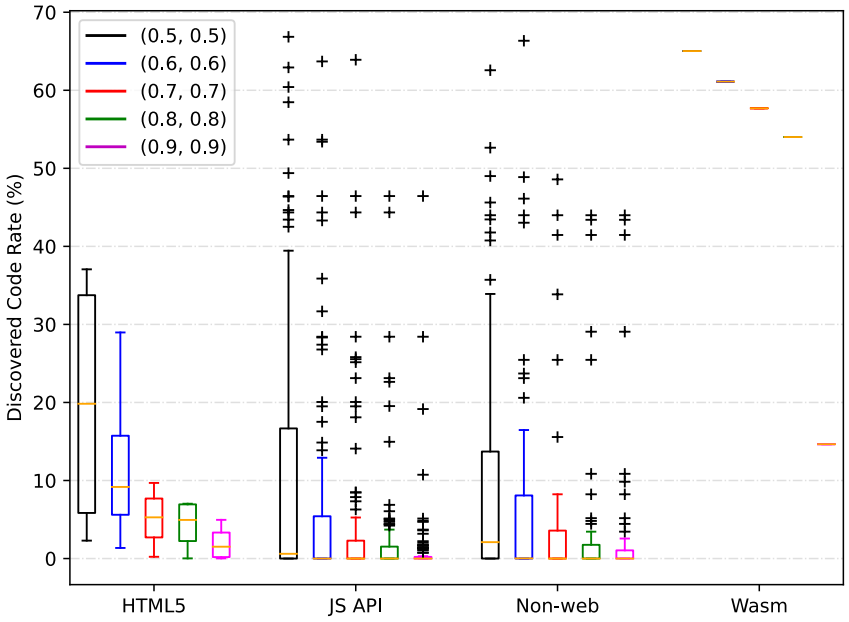


Figure 5.4: Breakdown of additional code discovery rates for each feature group across different relation vectors.

implies a moderate variance with outliers, but the medians consistently decrease at all four groups when raising those parameters because it means less code would be included for a feature, intolerant of fewer call invocations and dissimilar path names. For non-web

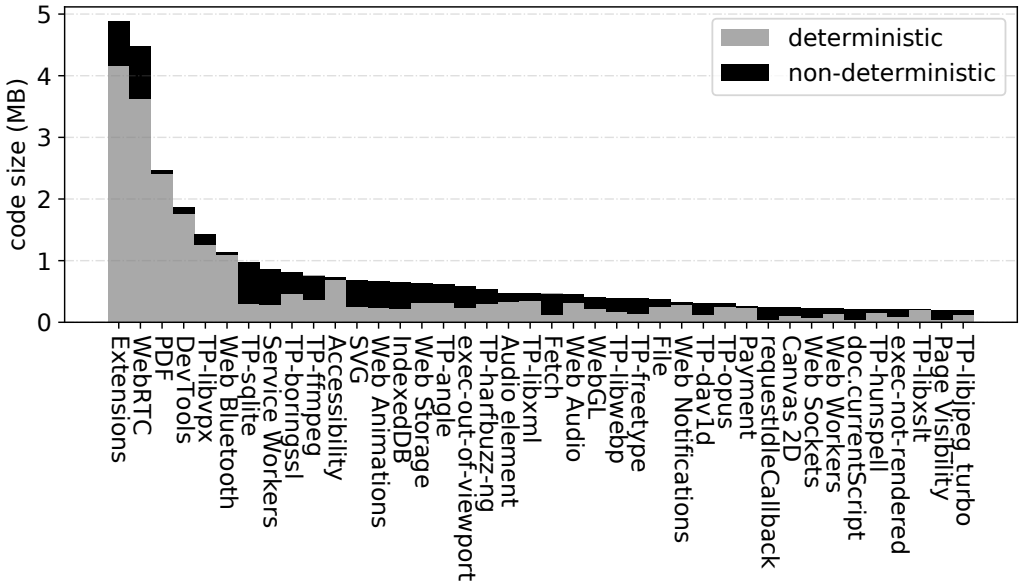


Figure 5.5: Ratio between non-deterministic code (dark bars on top) and the rest for the selected features. A prefix of TP_ represents a third-party component.

features, the median is close to zero because of many features from third-party libraries, which implies that those components have a minimal dependency. In our experiment, we set both hyperparameters (r_c and r_s) to 0.7, resulting in a 9.3% code increase (41.1 → 44.9 MB) on average, each of which breaks down into 4.9%, 8.6%, 5.0%, and 57.7% for HTML5, JSAPI, Non-Web, and Wasm, respectively. We discuss how to select the best hyperparameters for SLIMIUM in Section §5.3.3.

5.3.2 Non-deterministic Paths Discovery with Webpage Profiling

To identify non-deterministic code paths, we performed webpage profiling for the Top 1000 Alexa websites in an automated fashion: opening a main page of each site in Chromium, waiting for 5 seconds to load, exiting, and repeating until no more exercised functions are found via a differential analysis. Our empirical results show that it requires continuous visits of 172 times on average.

Figure Figure 5.5 illustrates non-deterministic portions of whole code from the top 40 debloatable features in size. The rate varies depending on each feature. On the one hand,

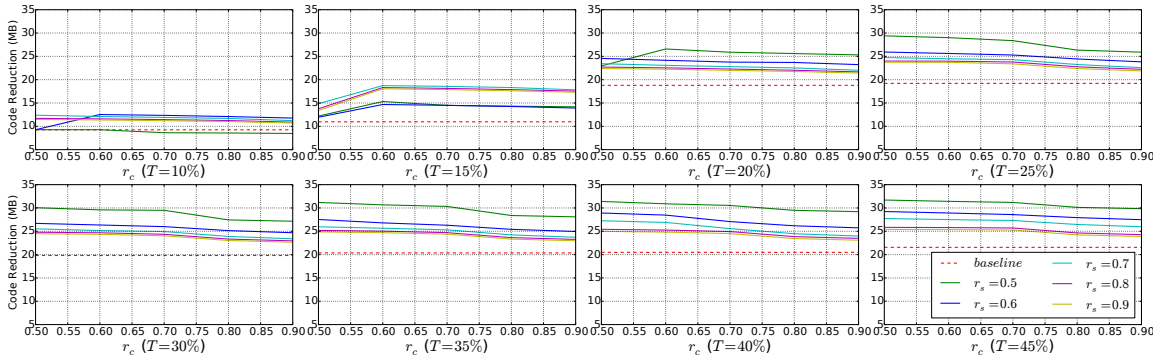


Figure 5.6: Average code reduction with a combination of different thresholds (r_c: call invocation, r_s:name similarity, T: code coverage rate) when loading the front page of the Top 1000 Alexa websites. The `baseline` represents the size of code reduction based on the initial feature-code map before applying Algorithm Algorithm 3 in Appendix.

network (`Service Workers`, `Fetch`), local cache (`third_party_sqlite`, `IndexedDB`) and animation features (`SVG`, `Web Animations`) trigger a substantial portion of non-deterministic code, mostly because they are designed for flexible behaviors (i.e., networking, caching, animation). For example, `Service Workers` acts as proxy servers, which heavily relies on network status and caching mechanism, and `Fetch` fetches resources across the network. In a similar vein, `third_party_sqlite` and `IndexedDB` store data on a local machine for future services. Interestingly, we observe that non-deterministic code rates of `SVG` and `Web Animations` are also high because, in general, advertisements employ those features to trace any changes whenever a page is reloaded. On the other hand, features such as `PDF`, `Web Bluetooth`, and `Accessibility` maintain a low non-deterministic code rate (i.e., opening the same PDF document).

5.3.3 Hyperparameter Tuning

In this section, we explore four tunable hyperparameters (thresholds) for SLIMIUM: i) two relation vector components: call invocation (r_c) and name similarity (r_s) (Section §5.2.2), ii) code coverage rate (T), and iii) sliding window size, that is, the length of revisiting times that no new function has been exercised. We empirically set up the sliding window as 10

described in Section §5.2.3, hence here focuses on the other three thresholds.

In particular, T is an important factor that determines whether a feature can be a candidate to be further eliminated based on the portion of exercised code. This is because i) part of feature code may contain initialization or configuration for another and ii) a small fraction of feature code may be exercised for a certain functionality. Under such cases, we safely maintain the exercised functions of that feature instead of removing the whole feature code. To summarize, we keep entire code (at the feature granularity) if code coverage rate is larger than T or exercised code alone (at the function granularity) otherwise.

Figure Figure 5.6 depicts the size of eliminated code on average by loading the front page of the Top 1000 Alexa websites to explore the best hyperparameters, namely $\vec{R_F} = (r_c, r_s)$ and T, empirically. Each subfigure represents the size of code reduction (y-axis) depending on a different combination of r_c (x-axis) and r_s (line) where both r_c and r_s range from 0.5 to 0.9, and T is a fixed value (ranging from 10% to 45%). One insight is that a higher T value improves code reduction because unexercised code for more features has been removed; e.g., 11.9 MB code removal with (r_c, r_s, T) = (0.7, 0.7, 0.1) whereas 27.3 MB with (0.7, 0.7, 0.45). Another insight is that a lower pair of (r_c, r_s) often improves code reduction. For example, 30.1 MB code has been removed where T is 30% with (r_c, r_s) = (0.5, 0.5) while only 24.9 MB with (r_c, r_s) = (0.7, 0.7). However, sometimes a lower pair of (r_c, r_s) does not improve code reduction because of the dynamics of our code discovery process; e.g., the first three subfigures indicate that code reduction with 0.6 of r_c is not smaller than the one with 0.5 or 0.55.

However, higher code reduction may decrease reliability of a debloated variant because it increases the chance of erasing non-deterministic code. As it is significant to strike a balance between code elimination and reliable binary instrumentation, we finally choose (r_c, r_s, T) = (0.7, 0.7, 0.3) with the following three observations. First, there is little impact on code reduction increase when T reaches up to around 25%. Second, with a r_s fixed, code reduction decreases from $r_c = 0.7$ heavily (i.e., T is 25%, 30% or 35%). Third, with a r_c

fixed, code reduction slightly drops from $r_s = 0.7$. Exploration for a different combination of hyperparameters per feature looks also promising, which is open as part of our future research.

5.3.4 Chromium Debloating in Practice

Table 5.2: Code and CVE reduction across debloated variants of Chromium per each category (See Figure Figure 5.7 in detail).

Category	Websites	User Activities	Code Reduction Size (MB)	Code Reduction Rate (%)	Number of Removed CVEs
Airline	aa, delta, spirit, united	Login; search a flight; make a payment; cancel the flight, logout.	24.17	53.8	97
Email	gmail, icloud, outlook, yahoo	Login; read/delete/reply/send emails (with attachments); open attachments; logout.	23.75	52.9	97
Financial	americanexpress, chase, discover, paypal	Login; check a statement; pay a bill; transfer money; logout.	23.45	52.2	91
News	cnn, cnbc, nytimes, washingtonpost	Read breaking news; watch videos; search other news.	24.19	53.9	98
Remote Working	bluejeans, slack, webex, zoom	Schedule a meeting; video/audio chat; share a screen; end the meeting.	18.57	41.4	81
Shopping	amazon, costco, ebay, walmart	Login; track a previous order; look for a product; add it to the cart; checkout; logout.	24.33	54.2	98
Social Media	instagram, facebook, twitter, whatsapp	Login; follow/unfollow a person; write a post and comment; like a post; send a message; logout.	23.30	51.9	93
Sports	bleacherreport, espn, nfl, nba	Check news, schedules, stats, and players.	24.39	54.3	98
Travel	booking, expedia, priceline, tripadvisor	Login; search hotels; reserve a room; make a payment; logout.	24.16	53.8	97
Video	amazon, disneyplus, netflix, youtube	Search a keyword; play a video (forward/pause/resume); switch screen modes (normal/theatre/full) ; adjust a volume	24.18	53.9	93
All	–	–	17.43	38.8%	73

In this section, we choose 40 popular websites from 10 categories to thoroughly assess reliability and security benefits of our debloating framework in practice instead of just loading the main page of a website. Table Table 5.2 summarizes a series of user activities of each website and experimental results for both code and CVE reduction.

Code Reduction and Reliability

Code Reduction Table Table 5.2 shows empirical code reduction with a debloated Chromium that allows a limited number of websites per each category. It removes 53.1% (23.85 MB) of the whole feature code on average with a single exception of `Remote Working` category (41.4% removal) because it harnesses `WebRTC` (See Section §5.3.4 in detail). Likewise, a debloated mutation that supports all 40 websites removes 38.8% code (around 17.4MB) as the last line of Table Table 5.2.

Next, we evaluate code reduction relevant to security features including four major ones that are fundamentally important to a modern web environment: same origin policy

(SOP), content security policy (CSP), subresource integrity (SRI) and cross-origin resource sharing (CORS). Based on our manual profiling results, all 40 websites employ these four features where code coverage on average are 8.3%, 39.1%, 34.0% and 79.0% for SOP, CSP, SRI and CORS, respectively. Since SOP, CSP and SRI are not part of our feature set, SLIMIUM offers corresponding security features at all times. Although CORS is part of our feature-code map, we observe heavy use of this feature for the websites in our experiment (i.e., min/max code coverages are 60.5%/85.8%), thus SLIMIUM does not remove any code. We have other security related features in our feature-code map, such as `Credential Management`, `FIDO U2F`, and `Web Cryptography`. The corresponding code may be possibly removed because of low code coverage; for example, none of the 40 websites uses the feature of `FIDO U2F` based on our profiling results. In this case, if a website would trigger any code for a security feature removed by SLIMIUM, a debloated Chromium variant would throw an illegal instruction exception and stop loading a page rather than allow one to visit a website without that security feature.

Reliability We have repeated the same activities (from initial webpage profiling) using different mutations, resulting in *flawless browsing for all cases without any crash*. As a case study, we investigate three websites in a `Remote Working` category that offer their services with native applications on top of a Chromium engine. Both `Slack` and `Bluejeans` are built with an Electron framework [71] (embedded Chromium and *Node.js*), containing 109.4 MB and 111.6 MB code, respectively, where `Zoom` contains 99.6 MB. Compared to those applications, our debloated version for `Remote Working` sorely contains 91.4 MB (up to 18.1% code reduction) that maintains every needed functionality. Note that `Webex` has been ruled out because it runs on Java Virtual Machine.

With the different debloated versions of Chromium, we were able to visit all 40 websites flawlessly thanks to non-deterministic code identification and appropriate hyperparameter selection ((r_c, r_s, T) = (0.7, 0.7, 0.3)). However, theoretically it is possible to encounter

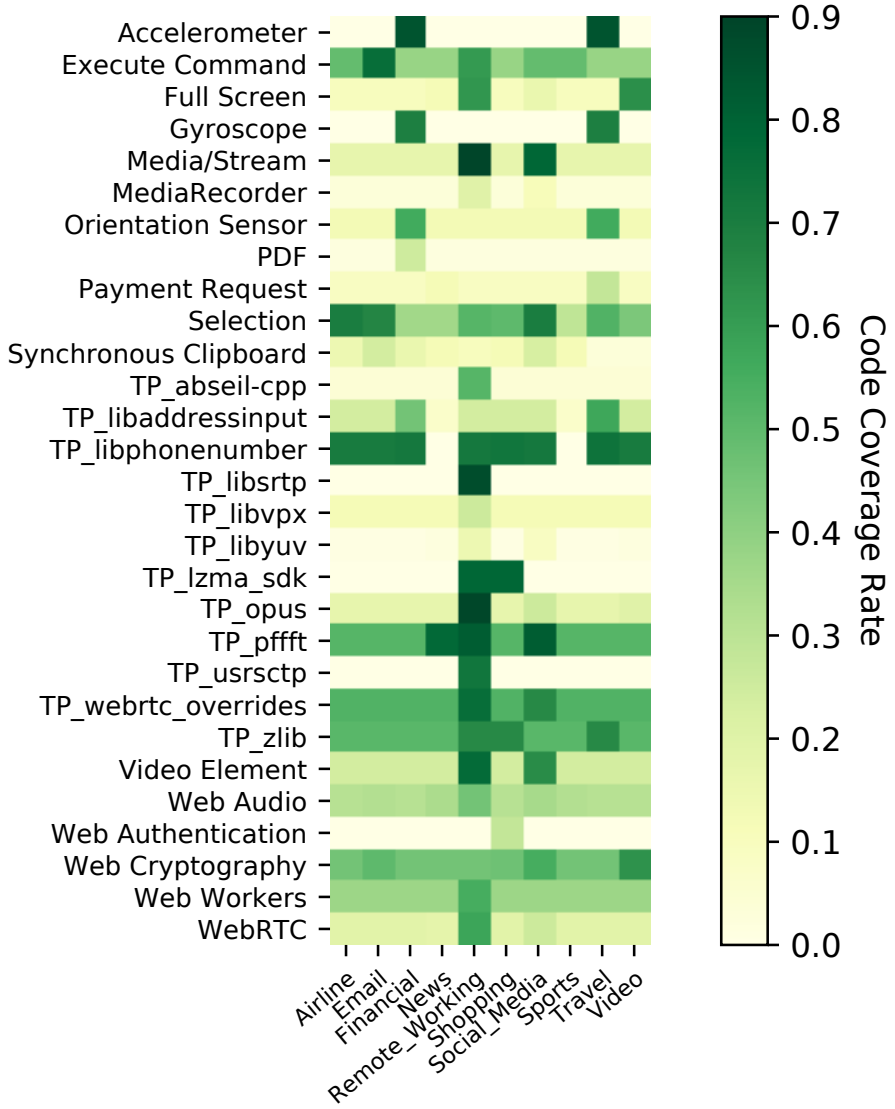


Figure 5.7: Code coverage rate of various features across different websites. A prefix of `TP_` represents a third-party component.

a false positive with a hyperparameter set of other choice. For example, by increasing T from 0.3 to 0.35, a debloated Chromium mutation would fail to load `Bluejeans` and `CNN` with the accidental removal of `Web Audio API`. Similarly, we observe additional failures of `Nytimes`, `Washingtonpost`, and `Whatsapp`, with the removal of `Web Workers` when T=0.4.

Feature Exploration

Figure Figure 5.7 depicts the heatmap for actual code coverage rates across different features per each category in Table Table 5.2 at a glance. Note that common features have been eliminated to show a distinct feature usage alone. The websites from the `Remote Working` group clearly adopt several unique Web features designed for Real-Time Communication (RTC) that are hardly seen from the others, including `WebRTC` and `Media Stream`, and the third party libraries (i.e., `webrtc_overrides`, `usrsctp`, `opus`, `libsrtp`). We also confirm a handful of interesting instances based on our activities as follows. A `PDF` feature has been rarely used but `Financial` because of opening PDF documents (i.e., bank statements). `Financial` and `Travel` have harnessed `Accelerometer`, `Gyroscope`, and `Orientation Sensor` for checking device orientation. `Remote Working` and `Video` sorely adopt `Full Screen` due to switching to a screen mode. Most websites employ the `libphonenumber` feature to maintain personal information with a login process whereas `News` and `Sports` do not. All the above examples explain that inner components in Chromium have been well-identified for the debloating purpose.

Security Benefits

To confirm the security benefits of our approach, we have collected 456 CVEs pertaining to Chromium from online resources [72, 73] and official (monthly) security updates from Google. We focus on the rest of the 364 CVEs that have been patched for the last two years (Some of them might be assigned in previous years), excluding 92 of them in case of i) no permission to access CVE information or ii) a vulnerability relevant to other platforms (i.e, iOS, Chrome OS). Although it may be less useful to count CVEs for a single version of Chromium because different versions are generally exposed to a different CVE set, we include them to evaluate the effectiveness of debloated mutations. It is noteworthy that we check out both vulnerable code and corresponding patches for mapping the CVEs to affected features.

Table Table B.1 in Appendix summarizes Chromium CVEs by 13 different vulnerability types and three severity levels that are associated with features for debloating. We adopt the severity level (i.e., High, Medium, Low) of each CVE calculated by the National Vulnerability Database (NVD) [73]. The most common vulnerability type is use-after-free (52), followed by overflow (46), and insufficient policy enforcement (43). Other vulnerability types include uninitialized memory, XSS (cross site scripting), and null dereference. Interested readers may refer to the full list of Chromium CVEs in Appendix Table Table B.3. Note that 153 (out of 364) CVEs are associated with 42 features in our feature-code map. From our experiments in Table Table 5.2, around 94 out of 153 potential CVEs (61.4%) have been removed on average when visiting the websites of our choice at each category. The number of CVEs that has been eliminated for the group of `Remote Working` is relatively low predominately due to RTC features. SLIMIUM successfully removes 73 CVEs (47.7%) across all 40 websites.

CHAPTER 6
CODE PARTITIONING APPROACH

I presented an approach for debloating Chromium at feature-level in Chapter §5. However, the approach relies on a feature-code map initially created with manual efforts, which is time-consuming and requires professional knowledge of the source code. In our experiment, the manual analysis took 40 working hours for an expert who is a maintainer of the Chromium's rendering engine (Blink [74]) to create the feature-code map at the first place. Therefore, it is challenging and time-consuming to extend the approach to debloat newer versions of Chromium with updates that change code heavily (e.g., adding new features, removing old features, reconstruct implementations of existing features, etc.), as well as to apply the technique to other large-scale and complex software. To tackle this problem, I will present DEPART, which is a framework that uses pure static analysis to automatically partitions a program into distinct groups implementing different features, which is later used for debloating.

6.1 Overview

Figure 6.1 shows an overview of DEPART for partitioning software's source code. DEPART consists of three main phases: i) type reference graph building, ii) relation construction, and iii) code partitioning.

Type Reference Graph Building In software that is developed with object-oriented programming (OOP) languages, an object is designed to have unique attributes and behaviors, and distinct features are implemented with different objects at most of the time. Thus, to distinguish the features implemented in a large-scale and complex software, we need to first identify the types (i.e., classes and structs in C/C++) defined in the source code and

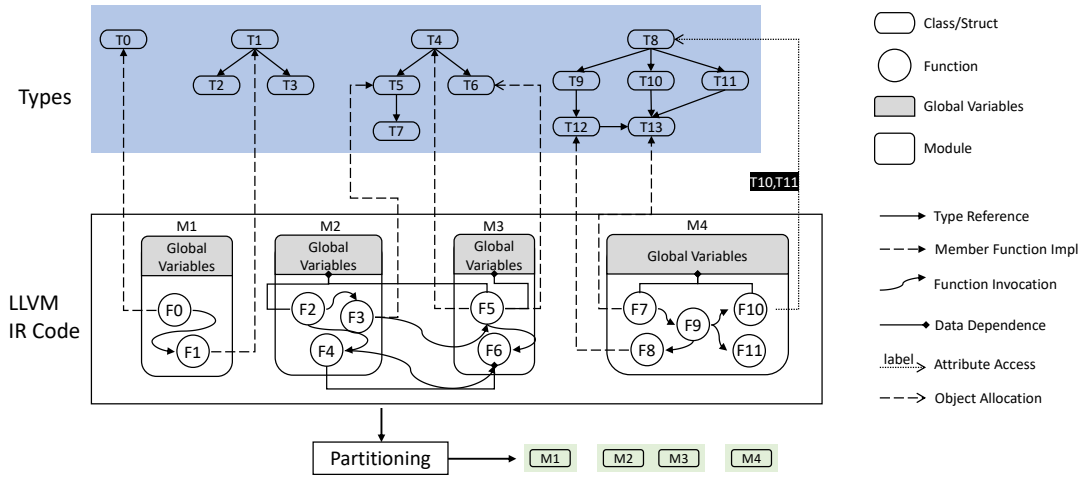


Figure 6.1: High-level overview of DEPART. It first takes in the software's LLVM IR bitcode files and perform static analysis to assign various relations among the types, global variables, and functions. After that, the PARTITIONING module applies rules to partition the code and types into distinct groups.

model their relationships. Since structs can be regarded as simpler classes without member functions, we will use types to indicate both classes and structs in the remaining of this chapter. DEPART iterates all the LLVM IR modules (each IR module is generated from a source code file) to extract all the types, and then it builds type reference graphs, in which each node represents a type and a directed edge exists between two type nodes when the source type refers to the destination type. The top part in Figure 6.1 shows that four type reference graphs are built from all the extracted types (e.g., type T0 does not refer to any other types, type T1 refers to types T2 and T3, etc.). We will discuss the details of building type reference graphs in §6.2.1.

Relation Construction In large-scale and complex software, a feature is commonly implemented across multiple source code files. Therefore, DEPART performs a set of static analyses to group IR modules implementing the same feature by assigning relations among types, functions, and global variables. There are three kinds of relations between a function and a type: (1) the function is one of the member functions of the type; (2) the function allocates objects of the type; (3) the function accesses a type's attributes. For example,

in Figure 6.1, the function F0 in module M1 is type T0's member function, the function F5 allocates objects of the type $T6$ and function F10 in module M4 accesses type T8's attributes. Note that the attribute access relation is labeled with all the accessed attributes' types (i.e., T0 and T11). Figure 6.1 also shows that the functions are related with either invocations (e.g., F0→F1) or data dependencies (e.g., F4→F6), and functions and global variables are related with data dependencies (e.g., F5→M3's GLOBAL VARIABLES). We describe the relations' definitions and the details about constructing these relations in §6.2.2.

Code Partitioning After assigning relations among the code (i.e., function), data (i.e., global variables) and types. DEPART's partitioning module defines rules and uses graph algorithms to group the IR modules that implement the same feature. The rules describe the conditions that the relations among types and IR modules must satisfy if they can be regarded as implementing the same feature. For example, in Figure 6.1, the partitioning module generates three groups: IR modules M2 and M3 are included into one group, while M1 and M2 are separated into two other groups. The definitions of the rules are described in §6.2.3.

6.2 Design

6.2.1 Type Reference Graph Building

Algorithm 2: Build the type reference graphs.

```
Result: typeRefGraphs
for classType : allClassTypes:
  | handleClassType(classType);
Function handleClass(classType)
  | for element : classType:
  |   | elemType = element.type;
  |   | if elemType→isPointerTy() || elemType→isAggregateTy():
  |   |   | elemType = getBaseType(elemType);          // Unwrap the type.
  |   | if elemType→isClassTy():
  |   |   | typeRefGraphs[classType].insert(elemType);
end
```

Algorithm 2 shows the pseudocode for building type reference graphs. It begins with iterating all the classes extracted from IR modules, and it calls the procedure *handleClass* to identify each class's type references. The procedure *handleClass* traversals the class's elements: for each element, it first obtains the element's type and then it calls the procedure *getBaseType* to unwrap the element's type if it is a pointer type or an aggregate type. After that, if the resulted type (i.e., *elemType*) is also a class type, then *elemType* is added into *classType*'s refer-to set. In other words, a directed edge is added between the source class node whose type is *classType* and the destination class node whose type is *elemType*.

6.2.2 Relation Construction

DEPART aims to partition the software to group IR modules into distinct sets implementing different features. Thus, the key challenge is to decide what IR modules should be regarded as implementing the same feature. Since each IR module consists of the data field (i.e., global variables) and code (i.e., functions); thus, DEPART performs static analysis on the global variables and functions to figure out the relations among IR modules and relies on the relations to make the decision. Also, DEPART leverages the interactions between code and types to help group IR modules.

Figure 6.1 shows that DEPART's static analysis assigns five kinds of relations among global variables, functions, and types. The relations are defined as follows:

- Member Function Implementation (**MFI**). This relation starts from a function F in a IR module and points to a class type C. The relation is assigned when F is a member function of C.

- Function Invocation (**FI**). This relation starts from a function $F1$ and points to another function $F2$. The relation is assigned when $F1$ invokes $F2$.

- Object Allocation (**OA**). This relation starts from a function F and points to a type C. The relation is assigned when F allocates objects of type C.

- Data Dependence (**DD**). This relation starts from a function $F1$ and points to another function $F2$ or points to a global variable GV. The relation is assigned between $F1$ and $F2$ when: (1) $F1$ invokes $F2$ and $F1$ writes the return value from $F2$; (2) $F2$ invokes $F1$ and $F1$ writes the parameters passed from $F2$. The relation is assigned between $F1$ and GV when $F1$ reads or writes GV.

- Attribute Access (**AA**). This relation starts from a function F and points to a type C, and the relation is labeled with $\{C_1, C_2, ..., C_n\}$. The relation is created when F accesses attributes of an object of type C and the accessed attributes' types are $\{C_1, C_2, ..., C_n\}$.

DEPART performs static analysis to assign the relations among types, functions, and global variables. Identifying a class's member functions is straightforward from checking the names because the member functions' names have a prefix string containing the class' name. For instance, if a class's name is STUDENT, then its member functions' names share the prefix "STUDENT::" in C++. However, to construct the other four relations: **FI**, **OA**, **DD** and **AA**, we need to analyze the IR modules' code. In particular, constructing **DD** relations is challenging because of the widely used pointers in C/C++ code. To address this challenge, DEPART proposes a novel static analysis by combining points-to analysis and taint analysis. I will show a motivation example to describe how we construct the relations (i.e., **FI**, **OA**, **DD** and **AA**) with static analysis.

Figure 6.2 shows the C++ code of our motivation example. The code defines one global variable (i.e., *g*) and four classes (i.e., *A*, *B*, *Base* and *Derived*), in which class *Derived* inherits class *Base* that defines a virtual function *foo*. Besides, the code defines a function *test*, which first creates an object of class *Derived* (i.e., *d*) and invokes its virtual function *foo*. After that, *test* sets the global variable *g* to *42* and assigns the global variable to *d*'s nested attribute accessed through $d \rightarrow b.a.v$.

Figure 6.3 shows the simplified LLVM IR code of the example, we omit the definitions of some global variables and functions, and skip part of the instructions to make the

```
int g = 0;
class A {
    int v;
};

class B {
    A a;
};

class Base {
    virtual void foo() {...}
};

class Derived : public Base {
    B b;
    void foo() {...}
};

void test() {
    Derived *d = new Derived();
    d->foo();
    g = 42;
    d->b.a.v = g;
}
```

Figure 6.2: Source code of the example for illustrating relation construction.

```
@g = 0

@Derived_VT = [
               null,
               @Derived_TI,
               @Derived.foo
              ]

void @Derived(%cls.Derived* %0)
{
 %2 = alloca %cls.Derived*
 store %0, %2
 %3 = load %2
 //call cls.Base's constructor
 ...
 %5 = cast %3 to void***
 %6 = GEP @Derived_VT[2]
 %7 = cast %6 to void**
 store %7, %5
}

void @test() {
 %1 = alloca %cls.Derived*
 %2 = call @new(i64 16)
 %3 = cast %2 to %cls.Derived*
 call @Derived(%3)
 store %3, %1
 %5 = load %1
 %6 = cast %5 to void (%cls.Derived*)***
 %7 = load %6
 %8 = load %7
 call void %8(%5)
 store 42, @g
 %9 = load @g
 %10 = load %1
 %11 = GEP %10.b
 %12 = GEP %11.a
 %13 = GEP %12.v
 store %9, %13
 ret void
}
```

Figure 6.3: LLVM IR code of the example for illustrating relation construction.

illustration clearer. Besides *g*, the code also shows the global variable *Derived_VT*, which defines the virtual table of class *Derived*. The virtual table is an array of three pointers: the first one is a null pointer; the second one points to a string representing the type information (i.e., *@Derived_TI*); the last one is a function pointer of *Derived*'s virtual function *foo* (i.e., *@Derived.foo*). We list the IR instructions of two functions, including *test* and class

Derived's constructor function @*Derived* that is automatically generated by the compiler.

FI To build *function invocation* relations between functions, DEPART directly extract the `Call` instruction's target function if it is a direct call; otherwise DEPART adopts existing techniques [75, 76, 77, 78, 79] to resolve the targets if it is a virtual function call. For the remaining indirect calls that are not virtual function calls, DEPART simply ignores them because we do not want to introduce any over-approximations to relate functions not supposed to be connected. In Figure 6.3, the function @*test* directly invokes functions @*new* and @*Derived*, and indirectly invokes the virtual function @*Derived.foo*.

OA To assign *object allocation* relations between a function and a type, DEPART analyzes the function's `Alloca` instructions to extract the allocated object's type. Note that if the type is a pointer type or aggregate type (e.g., ArrayType, VectorType), DEPART unwraps it and obtains its element type as the destination of the **OA** relation. In Figure 6.3, two **OA** relations will be created from the functions @*Derived* and @*test* to the class type *Derived*.

AA To create *attribute access* relation between a function and a type, DEPART analyzes the function's `GEP` instructions. A `GEP` instruction in LLVM [80] is used to get the address of an object's element, it like a high-level version of the `lea` instruction in x86. DEPART first gets a `GEP` instruction's base object's type and unwraps the attribute's type, then a **AA** relation labeled with the attribute type is assigned between the function and the base object's type. In Figure 6.3, the `GEP` instruction in the function @*Derived* will not introduce a **AA** relation because @*Derived_VT* is not a class/struct type. On the other hand, the first two `GEP` instructions create two **AA** relations: from @*test* to class type *Derived* with a label *"B"*, from @*test* to class type *B* with a label *"A"*, respectively. However, the last `GEP` instruction does not create a **AA** relation because the accessed attribute's type is not a class/struct type though the base object has the class type **A**.

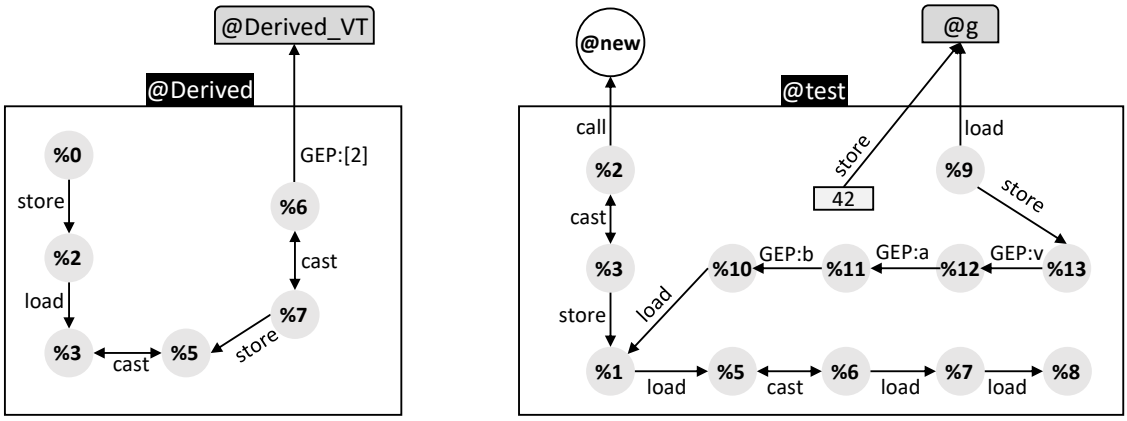


Figure 6.4: The points-to graphs for functions @*Derived* and @*test*.

DD To build the *data dependence* relations, analyzing a single type of instructions is not sufficient. Instead, besides instructions `Load` and `Store` that are used for data reading and writing, the other instructions whose operations affect the operands of `Load` and `Store` are also considered. Traditional data dependence analysis using backward data analysis is not effective enough to capture all the **DD** relations because of the widely spread pointers in C/C++. For example, to decide where the second `Store` instruction in function @*Derived* writes, the backward data analysis starts from the destination operand (i.e., *%5*) and traces all the way back to *%3*, which is loaded from *%2*. Because *%2* is a `Alloca` instruction that allocates a new address and points to nowhere, thus the analysis stops and fails to capture the destination address, which is *%0* stored to *%2* right after the `Alloca` instruction.

In order to address the limitation in backward data analysis, we propose a novel static analysis combining points-to analysis and taint analysis. DEPART first iterates a function's instructions and builds a points-to graph. Not liking the classic points-to analysis [81, 82] that resolves the constraints sooner or later, DEPART builds the graph embedding the points-to constraints and never resolves them. Figure 6.4 shows the points-to graphs of @*Derived* and @*test*. In a points-to graph, each node is a pointer value and the directed edge is labeled with the instruction type. In particular, the `GEP` edge's label also contains the attribute being accessed.

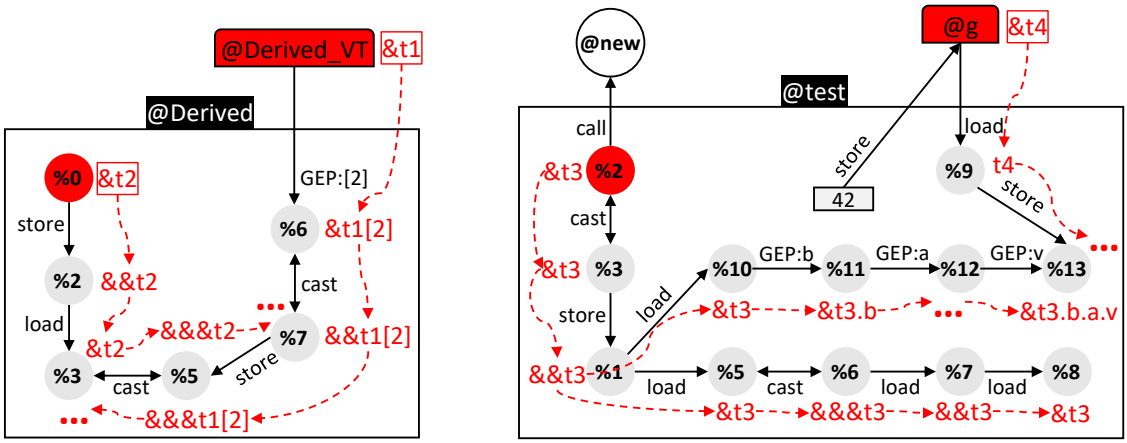


Figure 6.5: The taint propagations for functions @*Derived* and @*test*.

With the points-to graphs, before determining the memory pointed to by `Load`'s and `Store`'s operands, we first select the memory of our interest. Recall that the **DD** relation is assigned between two functions or between a function and a global variable, thus our goal is to figure out the code's read and write operations on three parts of the memory: (1) global variables; (2) the function's incoming arguments; (3) the return values of invocations. Next, instead of performing traversal on the points-to graphs to resolve each pointer's points-to set, which suffers from performance slowdown [81, 82]; DEPART performs taint analysis on the points-to graphs.

Figure 6.5 shows how the taint analysis works. DEPART first marks the pointers pointing to global variables, incoming arguments, and invocations' return values as taint sources. Each taint source is assigned a unique taint label and is appended one or multiple `&` at the beginning to indicate the number of the nested levels of the pointer. For example, global variables @*Derived_VT* and @*g* are tainted, *%0* is tainted because it is an incoming argument while *%2* is tainted because it is the return value of invoking function @*new*. All these taint source pointers are single-level pointers (i.e., a pointer that directly points to some memory); thus, their labels are appended with one `&` at the beginning (i.e., `&t1`, `&t2`, `&t3`, `&t4`).

For each tainted source, DEPART starts propagating its taint along the edges follow-

ing the instruction's semantic, as shown in Figure 6.5, which only presents the key taint propagation flows because of space limit.

- **`Store`** When the taint `[&]`^{n}`T` is being propagated from the source node to the destination node, DEPART taints the destination node with taint `[&]`$^{n+1}$`T` (e.g., *%0* $\rightarrow$ *%2* in function @*Derived*).

- **`Load`** When the taint `[&]`^{n}`T` is being propagated from the source node to the destination node, DEPART taints the destination node with taint `[&]`$^{n-1}$`T` if $n > 0$ (e.g. *%1* $\rightarrow$ *%5* in function @*test*); otherwise, taint is not propagated to the destination node.

- **`Cast`** When the taint is being propagated between source node and destination node, DEPART first checks the source type's and destination type's levels of the pointers. If they are same, DEPART simply copy the taint from one node to another; otherwise DEPART propagates the taint by appending or removing `&`^{n} at the beginning of the original taint, in which n is the distance between the levels of pointers. For example, for the case *%2* $\leftrightarrow$ *%3* in function @*test*, the instruction is casted from `i8*` to `%cls.Derived*`, both of which are single-level pointers; thus the taint is just copied between *%2* and *%3*. However, for the case *%5* $\leftrightarrow$ *%6* in function @*test*, the instruction is cast from `%cls.Derived*` to `void (%cls.Derived8)***`, in which the source type is a single-level pointer while the destination type is a three-levels pointer; thus the distance is two and the taint `&t3` in *%5* becomes `&&&t3` in *%6*.

- **`GEP`** For **GEP** instruction, the taint propagation strategy between the source node and destination node follows the semantic of *assignment*. The insight is that when an object is tainted with label `T`, its attributes are also tainted with labels in the format of `T.attr`. Therefore, the taint of the source node should be appended with the attribute's name and copied to the destination node (e.g., @*Derived_VT* $\rightarrow$ *%6*, *%10* $\rightarrow$ *%11*).

After finishing the taint propagation, DEPART checks `Load` and `Store` instructions' operands' taint sets to determine the memory read or written. For a `Load` instruction, DEPART checks its source node's taint set and removes all the & in any taints to extract the memory that is read. Take the `Load` instruction in @*Derived* as an example, *%2* has the taint `&&t2`; thus, memory tainted with `t2` is regarded as being read by the code, which indicates the memory of incoming argument *%0*. For a `Store` instruction, DEPART cross-checks the source node's and the destination node's taint sets to extract the memory addresses it reads and writes. Particularly, we filter out the read-write pairs originates from the same taint (e.g., `&&t1[2]` in *%7* and `&&&t1[2]` in *%5*). Therefore, by checking the `Store` instructions in functions @*Derived* and @*test*, we find that @*Derived* writes `t1[2]` to `t2` and @*test* writes `t4` to `t3.b.a.v`.

6.2.3 Code Partitioning

Once the relation construction is complete, DEPART first defines rules to build graphs connecting IR modules and the types, and then employs classic graph algorithms for finding connected components (CCs) [83] or strongly connected components (SCCs) [84] to group IR modules and types. To this end, we introduce some definitions that will be used in describing the code partitioning rules.

Definition 1 (Type Component)**.** A type component TC is a set of types, in which any two types are connected in type reference graphs, as we have discussed in §6.2.1.

Definition 2 (Module)**.** A module M contains a set of functions and global variables: $M = (F_s, GV_s)$ ($F_s = \{F_1, F_2, ..., F_n\}$, $GV_s = \{GV_1, GV_2, ..., GV_m\}$).

Definition 3 (Group)**.** A group G comprises a set of IR modules and types: $G = (T_s, M_s)$ ($T_s = \{T_1, T_2, ..., T_n\}$, $M_s = \{M_1, M_2, ..., M_m\}$).

Definition 4 (Function Data-Dependence)**.** A function F_1 is data-dependent on a function F_2 if: (1) F_1 invokes F_2 and F_1 writes the return value; (2) F_1 is invoked by F_2 and F_1 writes

the incoming arguments; (3) there exists a global variable GV, F_1 reads GV and F_2 writes GV; (4) there exists a function F_3, F_1 is data-dependent on F_3 and F_3 is data-dependent on F_2.

Definition 5 (Function Control-Dependence)**.** A function F_1 is control-dependent on a function F_2 if: (1) F_1 invokes F_2; (2) there exists a function F_3, F_1 invokes F_3 and F_3 invokes F_2.

DEPART aims to do partitioning on a program's code and generates a set of groups ($\{G_1, G_2, ..., G_n\}$), and requires that each group implements a distinct feature. We will discuss the rules that DEPART uses to seek the modules and types that should be included in the same group.

Rule 1 . *Types belong to the same TC should be grouped into the same group.*

Rule 2 . *If an attribute with type T_1 is only accessed through an object with type T_2, then T_1 and T_2 should be in the same group.*

Rule 3 . *If a type T_1 is allocated by modules M_{s1}, a type T_2 is allocated by modules M_{s2} and M_{s1} is equal to M_{s2}, then T_1 and T_2 should be in the same group.*

Rule 4 . *If a module M has a function F, which is connected to a type T with **MFI** relation, then M and T should be grouped into the same group.*

Rule 5 . *For a set of functions $F_s = \{F_1, F_2, ..., F_n\}$ that belong to $M_s = \{M_1, M_2, ..., M_m\}$, if any two functions F_i and F_j in F_s are control-dependent on each other, then all the modules in M_s should be grouped into the same group.*

Rule 6 . *For a set of functions $F_s = \{F_1, F_2, ..., F_n\}$ that belong to $M_s = \{M_1, M_2, ..., M_m\}$, if any two functions F_i and F_j in F_s are data-dependent on each other, then all the modules in M_s should be grouped into the same group.*

Note that **Rule 1** to **Rule 3** are employed for grouping types while **Rule 4** to **Rule 6** are used for grouping the code and relating code to grouped types.

Table 6.1: Chromium's source code details.

	Type			Code		
	Class	**Struct**	**Total**	**Module**	**Function**	**Global Variable**
Number	53,044	15,868	68,912	26,451	622,105	125,997

Grouping Types In software developed with OOP languages, a feature's implementation usually defines a set of types that represent the objects involved in the feature. Therefore, the necessity of partitioning code into groups implementing distinct features is to group the types at first. DEPART expands the type reference graphs by adding edges required from applying **Rule 2-3**, and it uses use the graph algorithm to find the SCCs to group the types. After applying **Rule 1-3**, all the types that refer to each other, are uniquely referred and types whose objects are allocated in same modules, are all grouped.

Grouping Code Once the type grouping is completed, DEPART applies **Rule 4-6** to start code grouping on top of the type grouping's results. Note that applying **Rule 4-6** will cause merging of existing groups because different groups might contain code that is supposed to be in a same group.

6.3 Evaluation

In this section, we evaluate DEPART on Chromium's source code of version `77.0`, whose details are list in Table 6.1. In particular, we assess from the following two perspectives:

- **Effectiveness Of Code Partitioning.** How many types and how much code can be partitioned into distinct groups? What are the types and code distribution over groups? §6.3.1
- **Comparison With Manual Analysis.** Does DEPART achieve benefits when comparing with partitioning code with manual analysis? When applying DEPART to debloat Chromium, does it identify extra features not included by SLIMIUM? Does DEPART remove more code than SLIMIUM? How long does DEPART take to partition

Chromium? §6.3.2

6.3.1 Effectiveness of Code Partitioning

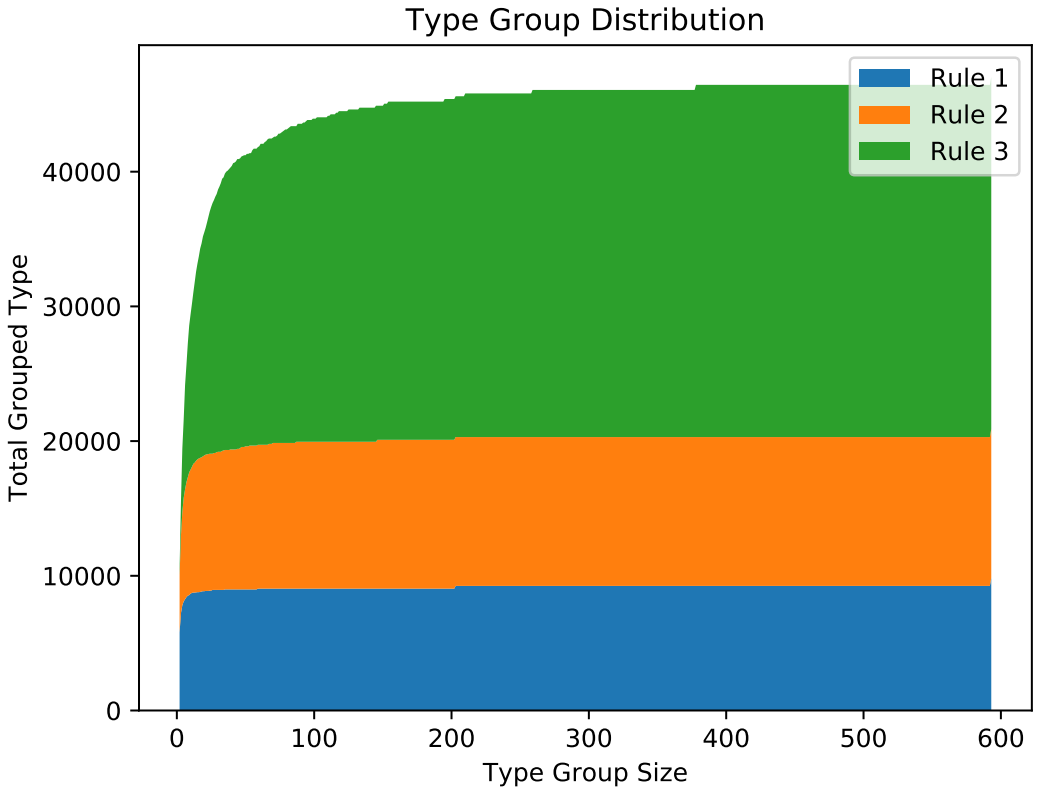
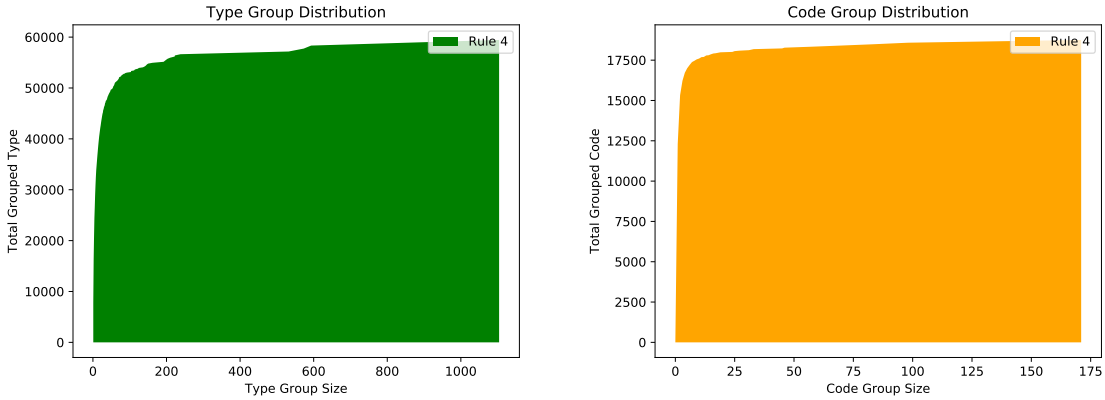


Figure 6.6: Apply **Rule 1-3** to group types.

Figure 6.6 shows the results of applying **Rule 1-3** to group class types. The x-axis is the type group's size starting from two while the y-axis is the accumulation of the grouped types. As we can see: (1) applying **Rule 1** groups 9,843 types into 3,680 groups, in which most groups share the sizes of two (i.e., 2,902 groups) and three (i.e., 475 groups); (2) applying **Rule 2** groups 20,892 types into 7,045 groups, in which most groups share the size between two and six (i.e., 6795 groups); (3) applying **Rule 3** groups 47,040 types into 10,088 groups whose sizes are mostly less than 30. Note that the biggest group contains 593 types, which represents the instruction operators defined in the JavaScript runtime `v8`'s source code file *machine-operator.cc*.



(a) Grouped types after applying **Rule 4**.

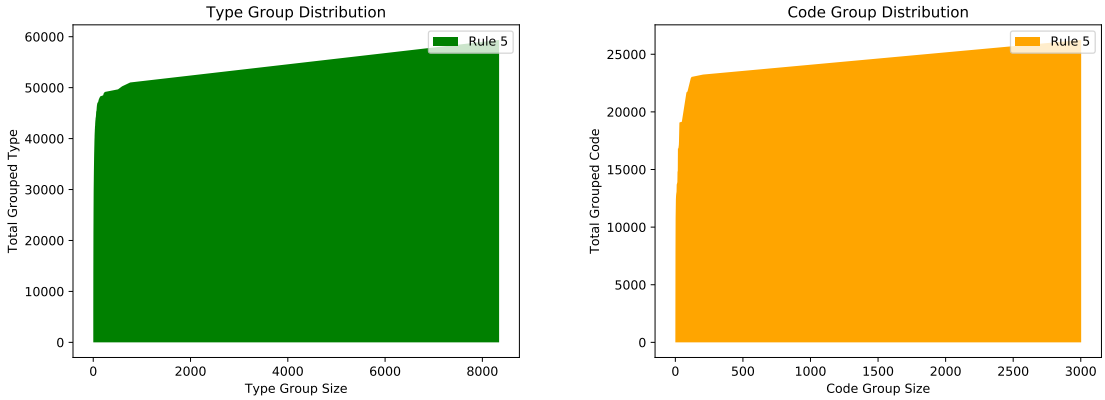
(b) Grouped code after applying **Rule 4**

Figure 6.7: Apply **Rule 4** to connect types and their member functions' implementations.

Figure 6.7 shows the grouped types and code after applying **Rule 4**, which basically connects types to code implementing the member functions and merges the type groups when their implementations share same code. Note that type groups with size one can also be grouped with code. In total, 59,441 types and 18,753 modules are grouped into 16,799 groups, in which some of the groups contain only types or code. In particular, Figure 6.7b shows that there are 122,97 groups containing only one code module while Figure 6.7a shows that there are 7,777 groups containing only one type, which means that it is common that multiple classes are defined and implemented in a same source code file. Note that the biggest group has 1,104 types and 171 code modules mainly from the JavaScript engine `v8` and the rendering engine `blink`.

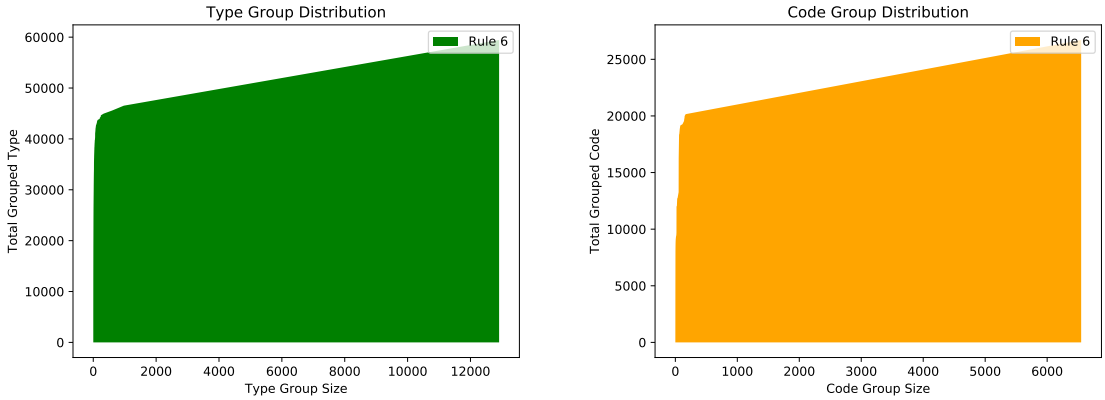
Figure 6.8 shows the grouped types and code after applying **Rule 5**, which merges groups with code control-dependent on each other. Comparing with the results from Figure 6.7, we find that the number of groups containing only one module decreases a lot (i.e., from 12,297 to 7,767) but still most of the groups have code from two to 10 modules. Note that the biggest group has 8,344 types and 3,003 modules. We investigated this group and figured out that this group mainly contains code across the UI of the browser, the rendering engine.

Figure 6.9 shows the grouped types and code after applying **Rule 6**, which merges



(a) Grouped types after applying **Rule 5**. (b) Grouped code after applying **Rule 5**

Figure 6.8: Apply **Rule 5** to group code control-dependent on each other.



(a) Grouped types after applying **Rule 6**. (b) Grouped code after applying **Rule 6**

Figure 6.9: Apply **Rule 6** to group code data-dependent on each other.

groups with code data-dependent on each other. Comparing with the results from Figure 6.8, we find that the number of groups containing only one module decreases a lot (i.e., from 7,767 to 5,123) and most of the groups have code from two to 100 modules, which means some of the small groups are merged because of data dependence relations. Note that the biggest group has 12,911 types and 6,549 modules, which is extended from the biggest group in Figure 6.8 by including more type and code related to the rendering engine, JavaScript engine and some code from the third-party library `Skia` that is used for graphics operations.

At this point, we have applied all the rules to group Chromium's code and types and

we get 5,123 groups containing only one module, 1,453 groups containing two modules, 165 modules containing three modules, etc., and one group containing 6,549 modules. We filter the biggest group and select the remaining 6,996 groups as the identified features, including 18,157 code modules and 33,737 types.

6.3.2 Comparison With Manual Analysis

The code size of the 6,996 groups is 62.04 MB (56.4%) of the entire Chromium binary, which achieves 38.2% increase over the feature-code map (44.9 MB) used in §5.3.1.

Feature Identification

Extra Identified Features 6.3.1
Printer/Cloud Printing, Debug, Json Parser, Media Galleries, Music Manager, Auto Fill, Download Manager, Signing/Account, Spell Checker, Kids Management/Supervised User, Safe Browsing, Task Manager, Translator, Feature Engagement, Headless Mode, Themes, Task Manager, etc.

Comparing DEPART with SLIMIUM regarding feature identification is challenging. The reason is twofold: firstly, a feature in SLIMIUM usually contains larger code size because the manual analysis assumes that code under same directory belongs to the same feature (e.g., code under directory `third_party/pdfium`); secondly, DEPART does not tell the features' names while each feature in SLIMIUM is assigned a name manually. Therefore, we first check the intersections between each group from DEPART and each feature's code of SLIMIUM, if there are intersections, then the group from DEPART is regarded as a sub-feature of the feature identified in SLIMIUM. By doing this, we find that each feature from SLIMIUM is divided into 28 sub-features in DEPART. In particular, the feature `PDF` is divided into 96 dub-features and the feature `WebRTC` is divided into 92 sub-features. Second, to find out the extra features identified by DEPART, we manually investigate the groups sharing no code intersections with any features in SLIMIUM and try to figure out

feature name from the source code file names and the types included in the groups. The above text box lists part of the features identified by DEPART.

Code Reduction

Table 6.2: Code reduction across debloated variants of Chromium per each category.

Category	Websites	SLIMIUM Size (MB)	DEPART (164 Features) Size (MB)	DEPART (All Features) Size (MB)
Airline	aa, delta, spirit, united	24.17	23.82(-1.45%)	32.45(+34.26%)
Email	gmail, icloud, outlook, yahoo	23.75	23.18(-2.40%)	31.89(+34.27%)
Financial	americanexpress, chase, discover, paypal	23.45	21.92(-6.52%)	31.67(+35.05%)
News	cnn, cnbc, nytimes, washingtonpost	24.19	22.42(-7.32%)	32.50(+34.35%)
Remote Working	bluejeans, slack, webex, zoom	18.57	17.18(-7.49%)	27.53(+48.25%)
Shopping	amazon, costco, ebay, walmart	24.33	22.84(-6.12%)	32.90(+35.22%)
Social Media	instagram, facebook, twitter, whatsapp	23.30	21.95(-5.70%)	31.76(+35.92%)
Sports	bleacherreport, espn, nfl, nba	24.39	21.87(-10.33%)	31.98(+31.12%)
Travel	booking, expedia, priceline, tripadvisor	24.16	22.77(-5.75%)	32.91(+36.22%)
Video	amazon, disneyplus, netflix, youtube	24.18	22.88(-5.38%)	32.49(34.37+%)
All	–	17.43	16.72(-4.07%)	26.78(+53.64%)

To determine the code reduction that DEPART can achieve, we follow the experiment performed in §5.3.4 to use the code partitioning results to debloat Chromium for 40 popular websites. Table 6.2 shows the code reduction results, in which the third column shows the code reduction of SLIMIUM, the fourth column shows the code reduction of DEPART when applying debloating the groups sharing code intersections with SLIMIUM's 164 features, and the last column shows DEPART's code reduction when applying debloating all the groups resulted from partitioning.

Note that DEPART remove less code than SLIMIUM when debloating the groups sharing code with the 164 features identified in SLIMIUM. The reason is that: in SLIMIUM, a feature's code size is larger, thus when using the code coverage threshold (i.e., 35%) to decide removing a feature or not, the feature is decided to be removed and all the feature's unexecuted code will be reduced; however, when the feature is divided into multiple smaller sub-features in DEPART and using the same code coverage threshold usually decides to keep the sub-features. On the other hand, Table 6.2 shows that DEPART removes above

Table 6.3: DEPART's peformance.

	Time (min:second)	Memory (GB)
Relation Construction	19:33	33.89
Code Partitioning	3:21	6.12

31% code than SLIMIUM when applying to all the partitioned groups, which makes sense as DEPART identifies more features than SLIMIUM.

Performance

Table 6.3 shows the time and memory performance of running DEPART. The relation construction is resource-consuming because it parses all the 26,451 code modules, performs static analysis on the code and constructs the relations among types and functions. Besides that, the code partitionning phase only takes three minutes and 21 seconds, which is much faster than the manual analysis in SLIMIUM (i.e., 40 hours).

CHAPTER 7
REFLECTIONS

In this chapter, I will first discuss the limitations of the proposed work in this thesis. Later, I will discuss some of the future directions that this dissertation opens up.

7.1 Limitations

This section discusses the limitations of the proposed work above from the perspectives of high-level feature extraction, memory overhead, and fault handling.

7.1.1 High-level Feature Extraction

We have seen that DEPART can do code partitioning on a program and divide the code into a set of groups implementing distinct features. However, the generated groups mostly contain small code with less than 10 code modules, which implement sub-features belong to a high-level feature. This issue is caused by the limitations of the rules that DEPART uses to group types and code. The rules defined in DEPART either heavily rely on relation recursion to completely avoid grouping types and code that cannot be proved 100% related (e.g., type reference recursion, control-dependence recursion, data-dependence recursion) or only include types or code are always bonded by the same operations (e.g., object allocation in same modules, exclusively accessing attributes).

Not liking the feature-code map generated from manual analysis in SLIMIUM, in which each feature has a name; the partitioned code resulted from DEPART do not have names, rendering the extracted features cannot be understood at high level.

7.1.2 Memory Overhead

Besides reducing attack surface, another benefit is supposed to achieve from debloating is saving memory usages at runtime, especially on IoT devices that suffer from the limited resource problem. However, none of the three proposed systems achieves this goal. SLIMIUM and DEPART rewrites the binary through marking the unneeded code with illegal instructions, resulting the same-sized binaries. More seriously, RAZOR produces debloated binaries using more memory than the originals because it creates a new code section in the binary; and the old code section is only marked read-only instead of being removed because of supporting some corner cases.

7.1.3 Fault Handling

Like all the existing debloating systems, our proposed work also has the limitation of handling the situations in which the removed code is triggered. RAZOR, SLIMIUM and DEPART choose to exit the running process once this scenario happens, which is not only user-unfriendly but also possible to cause security issues. For example, users' data might be lost or user' privacy can be leaked because the running process for data backup or data encryption is killed.

7.2 Future Work

This section discusses some future directions that this dissertation opens up.

Improving static analysis. We have seen that employing static analysis in debloating systems helps generating debloated software with higher robustness. RAZOR syntactically analyzes the binary instructions to infer code similar with executed code, which generates binaries that support running extra inputs other than the inputs used for training. SLIMIUM uses static analysis to extend the feature-code map created manually and it helps reduce more code. DEPART employs static analysis to automatically partition a program into

groups implementing distinct features, which saves the manual efforts for building feature-code map. However, as we have discussed in §7.1.1, the static analysis still has limitations causing the failure of generating high-level features. We believe that the static analysis can be improved by constructing more relations with higher semantics (e.g., detecting common patterns of invoking member functions, chaining memory reads/writes, etc.). Furthermore, we will improve the static analysis to define rules matching design patterns used in software industry. Finally, exploring new static analysis involved with probabilistic approaches is also believed to be beneficial for code partitioning.

Using NLP. To extract high-level features that can be understood by users, simply partitioning the code is not enough as assigning feature names to the code groups is necessary. We believe that applying NLP techniques on the grouped code and types can help extracting the feature names automatically.

Customizing compiling toolchains. Customizing the compiling toolchain (i.e., compilers, linkers) is a powerful approach for instrumenting programs. To address the memory overhead limitation discussed in §7.1.2, one possible solution is to modify the compiler to generate shadow copies of the code that only contain empty functions and customizes the linker to link the shadow code as required. Take a further step, we can also customize the compiling toolchain to insert fault handlers in the program to gently handle the situations when the removed code is triggered. For example, a fault handler that asks users to make the choices between exiting the process and restoring the removed code would be acceptable.

CHAPTER 8
CONCLUSION

Previous debloating systems ask end-users to provide a set of sample inputs to demonstrate how they will use the software and generate a debloated version of the software that only contains the code triggered from running the sample inputs. Unfortunately, software debloated by this approach only supports running given inputs, presenting an unusable notion of debloating: if the debloated software only needs to support a fixed set of inputs, the debloating process is as simple as synthesizing a map from the input to the observed output. We call this *Over-debloating Problem*. This dissertation focuses on removing software's unneeded code while providing high robustness for debloated software to run more inputs sharing the same functionalities with the given inputs, with approaches either using heuristics, feature-code map, or code partitioning.

First, the thesis presents RAZOR, which first collects executed code for running the software on the given inputs and then uses heuristics to infer non-executed code related to the given inputs. In the end, RAZOR rewrites the software to keep not only the executed code but also the inferred code, which makes the debloated software support running other inputs besides the given ones. However, in RAZOR, the heuristics are syntax-based and can only infer a limited set of related code, which fails on debloating large-scale and complex software such as web browsers.

Later, the thesis presents SLIMIUM, which uses a feature-code map to debloat the web browser Chromium at feature-level. In SLIMIUM, the feature-code map is initially created from manual analysis and then it is expanded using static program analysis. However, relying on manual efforts to identify features and relevant code is time-consuming and difficult to be applied to other software.

Finally, the thesis presents DEPART, which provides a general approach to debloat

large-scale and complex software written with object-oriented programming (OOP) languages without any manual efforts. DEPART performs pure static analysis to automatically partitions a program into distinct groups implementing different features, which is later used for debloating. The key idea of DEPART is to relate the software's code and types (i.e., defined objects sharing unique behaviors in OOP) by analyzing the code's various operations. Based on the relations, we propose several rules to describe the conditions that should be satisfied for including types and code into a same group.

Appendices

APPENDIX A

RAZOR

Program	Training Set Size	Testing Set Size	Options
bzip2	10	30	`-c`
chown	6	17	`-h, -R`
date	22	33	`-date, -d, -rfc-3339, -utc`
grep	19	38	`-a, -n, -o, -v, -i, -w, -x`
gzip	10	30	`-c`
mkdir	12	24	`-m, -p`
rm	10	20	`-f, -r`
sort	12	28	`-r, -s, -u, -z`
tar	10	30	`-c, -f`
uniq	24	40	`-c, -d, -f, -i, -s, -u, -w`

Table A.1: Settings for evaluating PATHFINDER on the CHISEL benchmarks. We use the training set to debloat the binary, and run the generated code with the testing set. The last column is the options we pass to the binaries during training and testing.

APPENDIX B

SLIMIUM

Algorithm 3: Explore a pertinent object to a feature

```
Result: added_objects
in_nodes = feature.in_nodes;          // Obtain the incoming/outgoing nodes
out_nodes = feature.out_nodes;
r_c, r_s = 0.7;                                  // Initialize hyperparameters
for node : in_nodes:
    sum_c, sum_in_c, sum_in_s, in_num = 0;
    for edge : node.out_edges:
        (c, s) = edge.relation;
        sum_c += c;                                  // Sum up call invocations
        if edge.end_node in feature.nodes:
            sum_in_c += c;
            sum_in_s += s;                // Sum up hamming distance values
            in_num += 1;
    if sum_in_c / sum_c > r_c || sum_in_s / in_num > r_s:
        added_objects.add(node);
for node : out_nodes:
    sum_c, sum_in_c, sum_in_s, in_num = 0;
    for edge : node.in_edges:
        ...
        if edge.start_node in feature.nodes:
    ...     ...
```

Table B.1: Summary of Chromium CVEs and relevant unit features for debloating.

Vulnerability Type	High	Medium	Low	Total	Relevant Features
Bad cast	1	0	0	1	-
Bypass	3	22	2	27	document.{currentScript, domain}, Page Visibility, requestIdleCallback, Extensions, Service Workers, Video, and Web Audio
Disclosure	1	16	1	18	Extensions, Media Source, third_party_boringssl, Timing, Video, and Web Audio
Inappropriate implementation	0	14	1	15	DevTools, Extensions, and PDF
Incorrect security, handling, permissions	9	31	0	40	Extensions, DevTools, Navigator, Service Workers, URL, URL formatter, Web Assembly, WebRTC, and XMLHttpRequest
Insufficient policy enforcement	3	36	4	43	Canvas 2D, createImageBitmap, DevTools, Extensions, Payment, WebGL, Service Workers, Shared Web Workers, Page Visibility, requestIdleCallback, createImageBitmap, and document.{currentScript, and domain}
Insufficient validation	5	11	0	16	DevTools, IndexedDB, PDF, WebGL, and Web Assembly
Out of bound read	2	12	0	14	PDF, third_party_sqlite, and WebRTC
Out of bound write	3	5	0	8	PDF and Web Assembly
Overflow	14	32	0	46	Blob, Canvas 2D, Media Stream, PDF, Web Assembly, Web SQL, WebGL, WebGPU, WebRTC, and third_party_{angle, icu, and libxml}
Spoof	1	26	3	30	DevTools, Extensions, Full screen, Media Stream, and Web Bluetooth
Type Confusion	5	2	0	7	PDF, SVG, and WebRTC
Use after free	16	36	0	52	DevTools, Extensions, File, File System, IndexedDB, Media Capture, MediaRecorder, PDF, Payment, Web Assembly, Web Audio, Web MIDI, WebRTC, createImageBitmap, execution-while-out-of-viewport, and third_party_{libvpx, and libxml}
Others	8	34	5	47	Canvas 2D, createImageBitmap, DevTools, Directory selection, Extensions, Full screen, PDF, Service Workers, Web Assembly, Web Audio, WebRTC, and third_party_ffmpeg
Total	71	277	16	364	

Table B.2: Chromium features as a unit of debloating. The columns V, P, C, and E represent the number of CVEs, Feature Policy support, Chromium support, and Experimental flag support respectively (Yes: ●, No: ✗, Partial: ◗).

Class	Feature Name	Func.	Size (B)	V	P	C	E	Class	Feature Name	Func.	Size (B)	V	P	C	E
	Accessibility	3,018	774,862		✗	●	✗		Synchronous Clipboard	500	108,156		✗	◗	✗
HTML5	Canvas 2D	1,153	254,979	3	✗	●	✗	JS API	TextEncoder & TextDecoder	68	15,068		✗	●	✗
	SVG	3,884	710,004	1	✗	●	✗		Timing	634	142,366	1	✗	●	✗
	Video	518	98,402	2	✗	●	✗		Touch events	117	61,599		✗	●	✗
	WebGL	1,789	445,175	5	✗	●	✗		URL	82	19,462	1	✗	●	✗
	WebGPU	470	80,706	5	✗	●	✗		Vibration	104	14,600		●	●	✗
	AbortController & AbortSignal	33	5,187		✗	●	✗		Wake Lock	142	20,954		●	●	✗
JS API	Accelerometer	13	887		●	●	●		Web Animations	2,524	705,348		●	◗	✗
	Ambient Light Sensor	16	1,936		●	●	●		Web Audio	2,485	587,711	5	✗	●	✗
	Autoplay	65	13,763		●	●	✗		Web Authentication	28	1,972		●	●	●
	Background Sync	739	145,961		✗	●	●		Web Bluetooth	4,896	1,216,400	1	✗	●	●
	Base64	12	2,356		✗	●	✗		Web Cryptography	664	148,808		✗	●	✗
	Console	122	49,646		✗	●	✗		Web MIDI	339	56,585	1	●	●	✗
	Battery Status	123	13,889		●	●	✗		Web Notifications	1,539	342,313		✗	●	✗
	Beacon	13	2,071		✗	●	✗		Web Sockets	921	195,979		✗	●	✗
	BigInt	95	34,269		✗	●	✗		Web SQL	923	185,985	1	✗	●	✗
	Blob	850	205,670	1	✗	●	✗		Web Storage	3,177	698,731		✗	●	✗
	Broadcast Channel	116	17,948		✗	●	●		Web Workers	1,549	246,759		✗	●	✗
	Channel messaging	106	31,310		✗	●	✗		WebRTC	16,993	4,797,715	8	✗	●	●
	Constraint Validation	100	12,812		✗	●	✗		WebUSB	577	131,635		●	●	✗
	createImageBitmap()	613	116,031	4	✗	●	✗		WebVR	255	55,213		●	●	●
	Credential Management	224	57,680		✗	●	✗		WebXR	859	177,361		●	●	●
	Cross-Origin Resource Sharing	196	68,668		✗	●	✗		Window	325	73,359		✗	●	✗
	crypto.getRandomValues()	6	642		✗	●	✗		XMLHttpRequest	164	41,196	1	●	●	✗
	CSS.supports()	3	617		✗	●	✗		DevTools	10,273	2,889,315	15	✗	●	✗
	Custom Event	10	1,134		✗	●	✗	Non-web	Extensions	25,073	5,735,571	29	✗	●	✗
	Device Events	341	48,431		✗	◗	✗		NACL & PNACL	671	187,277		✗	●	✗
	Directory selection	324	53,452	1	✗	●	✗		PDF	8,253	2,601,303	26	✗	●	✗
	Do Not Track	9	1,403		✗	●	✗		third_party_abseil-cpp	30	3,274		✗	●	✗
	Document Object Model Range	89	37,579		✗	●	✗		third_party_angle	1,503	636,685	4	✗	●	✗
	document.currentScript	1,073	230,083	3	✗	●	✗		third_party_boringssl	2,142	824,106	1	✗	●	✗
	document.domain	929	209,395	3	●	●	✗		third_party_breakpad	109	49,047		✗	●	✗
	document.evaluate & XPath	335	92,237		✗	●	✗		third_party_brotli	42	34,158		✗	●	✗
	document.execCommand()	6	1,282		✗	●	✗		third_party_cacheinvalidation	891	187,681		✗	●	✗
	DOM Element	568	143,336		✗	●	✗		third_party_ced	46	50,842		✗	●	✗
	DOM Parsing and Serialization	457	111,963		✗	●	✗		third_party_cld_3	332	82,148		✗	●	✗
	Encrypted Media Extensions	374	67,794		●	●	✗		third_party_crc32c	3	3,209		✗	●	✗
	execution-while-not-rendered	1,029	217,103		●	●	✗		third_party_dav1d	422	316,415		✗	●	✗
	execution-while-out-of-viewport	2,621	587,719	1	●	●	✗		third_party_ffmpeg	1,453	795,694	1	✗	●	✗
	Feature Policy	152	53,400		✗	◗	●		third_party_flac	148	86,028		✗	●	✗
	Fetch	2,418	509,398		✗	●	✗		third_party_fontconfig	394	126,702		✗	●	✗
	FIDO U2F	1,456	467,216		✗	●	✗		third_party_freetype	776	400,392		✗	●	✗
	File	1,833	390,459	3	✗	●	✗		third_party_harfbuzz-ng	993	554,211		✗	●	✗
	Filesystem & FileWriter	947	169,129		✗	●	✗		third_party_hunspell	264	221,096		✗	●	✗
	Full Screen	196	42,364	5	●	●	●		third_party_iccjpeg	2	1,302		✗	●	✗
	Gamepad	709	143,215		✗	●	✗		third_party_icu	5,782	1,793,490	1	✗	●	✗
	Geolocation	235	41,809		●	●	✗		third_party_inspector_protocol	197	85,215		✗	●	✗
	Gyroscope	13	919		●	●	●		third_party_jsoncpp	79	37,373		✗	●	✗
	High Resolution Time	464	102,944		✗	●	✗		third_party_leveldatabase	608	180,080		✗	●	✗
	IndexedDB	3,053	861,111	2	✗	●	✗		third_party_libaddressinput	281	92,267		✗	●	✗
	Internationalization	73	45,243	1	✗	●	✗		third_party_libjingle_xmpp	399	87,405		✗	●	✗
	IntersectionObserver	191	44,669		✗	●	✗		third_party_libjpeg_turbo	325	196,271		✗	●	✗
	Intl.PluralRules	13	7,239		✗	●	✗		third_party_libphonenumber	235	85,665		✗	●	✗
	Magnetometer	13	919		●	●	●		third_party_libpng	231	99,653		✗	●	✗
	Media Capture	198	52,530	1	●	●	✗		third_party_libsrtp	121	34,315		✗	●	✗
	Media Recorder	275	67,833	1	✗	●	✗		third_party_libsync	1	163		✗	●	✗
	Media Source Extensions	374	67,794	1	✗	●	✗		third_party_libvpx	1,696	1,456,960	2	✗	●	✗
	Media Stream	83	14,057	2	✗	●	✗		third_party_libwebm	111	40,614		✗	●	✗
	Mutation Observer	179	34,649		✗	●	✗		third_party_libwebp	672	399,824		✗	●	✗
	Native Filesystem	433	104,131		✗	●	●		third_party_libxml	977	498,403	2	✗	●	✗
	navigator.hardwareConcurrency	1	19	1	✗	●	✗		third_party_libxslt	394	216,254		✗	●	✗
	Network Information	64	8,960		✗	●	✗		third_party_libyuv	455	172,021		✗	●	✗
	Orientation Sensor	54	7,314		✗	●	●		third_party_lzma_sdk	6	1,458		✗	●	✗
	oversized-images	59	18,273		●	●	✗		third_party_modp_b64	2	774		✗	●	✗
	Page Visibility	946	214,054	3	✗	●	✗		third_party_openscreen	1	67		✗	●	✗
	Payment	841	232,875	3	●	◗	✗		third_party_opus	304	313,488		✗	●	✗
	Permissions	494	119,978		✗	●	✗		third_party_ots	220	141,284		✗	●	✗
	Picture-in-Picture	225	35,203		●	●	✗		third_party_perfetto	720	183,312		✗	●	✗
	Pointer events	146	42,262		✗	●	✗		third_party_pffft	35	23,113		✗	●	✗
	Pointer Lock	14	1,786		✗	●	✗		third_party_re2	320	168,288		✗	●	✗
	Push	629	134,239		✗	●	✗		third_party_s2cellid	8	2,280		✗	●	✗
	requestAnimationFrame()	85	19,503		✗	●	✗		third_party_sfntly	1,350	114,962		✗	●	✗
	requestIdleCallback	1,175	260,757	3	✗	●	✗		third_party_smhasher	6	2,386		✗	●	✗
	Resize Observer	127	28,509		✗	●	✗		third_party_snappy	20	4,492		✗	●	✗
	Screen Orientation	170	26,174		✗	●	✗		third_party_sqlite	896	1,016,192	1	✗	●	✗
	Selection	511	156,461		✗	●	✗		third_party_tcmalloc	167	37,048		✗	●	✗
	Server-sent events	75	11,329		✗	●	✗		third_party_unrar	384	175,136		✗	●	✗
	Service Workers	4,518	1,049,970	4	✗	●	✗		third_party_usrsctp	397	310,183		✗	●	✗
	Shared Web Workers	419	79,337	1	✗	●	✗		third_party_webrtc_overrides	44	3,876		✗	●	✗
	Speech Recognition	843	229,729		✗	◗	✗		third_party_woff2	18	24,982		✗	●	✗
	Speech Synthesis	216	35,928		✗	●	✗		third_party_zlib	177	92,515		✗	●	✗
	Streams	698	126,798		✗	◗	✗		wasm	2,723	1,893,353		✗	●	✗

Table B.3: Chromium CVEs associated with our feature set. The severity column ranges from low(l), medium(▮) to high(■).

Features	Category	Sev.	CVE
Blob	Overflow	▮	2017-15416
Canvas 2D	Insufficient policy	■	2019-5766
		■	2019-5814
	Others	▮	2019-5787
	Overflow	■	2018-18338
	UAF	▮	2019-5758
DevTools	Inappropriate implementation	▮	2018-18344
	Incorrect security	l	2018-6112
		▮	2018-6112
		▮	2018-6139
	Insufficient policy	▮	2017-15393
		▮	2019-5768
	Insufficient validation	■	2018-6101
		▮	2018-6039
		▮	2018-6046
	Others	▮	2018-6152
	UAF	■	2018-6111
		▮	2018-6111
Directory selection	Others	▮	2018-6095
document.*	Bypass	▮	2019-5799
		▮	2019-5800
	Insufficient policy	▮	2018-18350
	UAF	■	2019-5759
Extensions	Bypass	▮	2018-6070
		▮	2018-6089
	Disclosure	■	2018-6179
	Inappropriate implementation	■	2018-20065
	Incorrect security	■	2018-16064
		■	2019-5793
		▮	2017-15420
		▮	2018-6110
	Insufficient policy	■	2019-13754
		l	2018-6045
		▮	2017-15391
		▮	2017-15394
		▮	2018-6035
		▮	2019-13755
		▮	2019-5778
	Others	■	2019-5838
		▮	2018-16086
		▮	2018-6121
		▮	2018-6138
		▮	2018-6176
		▮	2019-5796
	Spoof	▮	2019-13691
	UAF	■	2018-20066
		■	2018-6054
		▮	2018-20066
		▮	2019-5878
Extensions, DevTools	Incorrect security	▮	2018-6140
	Others	▮	2018-16081
	Spoof	▮	2018-6178
File	UAF	▮	2018-6123
		▮	2019-5786
		▮	2019-5788
File System		▮	2019-5872
Full screen	Others	▮	2018-17471
		▮	2018-17476
	Spoof	▮	2017-15386
		▮	2018-16080
		▮	2018-6096
IndexedDB	Insufficient validation	▮	2019-5773
	UAF	▮	2019-13693
Media Capture		■	2017-15395
Media Source Extensions	Disclosure	■	2018-16072
Media Stream	Overflow	▮	2019-5824
	Spoof	▮	2018-6103
MediaRecorder	UAF	■	2018-18340
Navigator	Incorrect security	▮	2018-6041
Payment	Insufficient policy	▮	2018-20071
		▮	2019-13763
	UAF	▮	2019-5828
PDF	Inappropriate implementation	▮	2018-20065
	Insufficient validation	■	2016-10403
	OOB read	▮	2018-16076
	OOB write	■	2017-5133
		▮	2018-6144
	Others	■	2019-13679
PDF	Overflow	l	2018-6120
		▮	2017-15408
		▮	2018-17461
		▮	2018-17469
		▮	2019-5792
		▮	2019-5795
		▮	2019-5820
		▮	2019-5821
	Type Confusion	▮	2018-6170
	UAF	■	2017-15410
		■	2017-5127
		■	2018-18336
		■	2019-5762
		▮	2017-15411
		▮	2017-5126
		▮	2018-6088
		▮	2019-5756
		▮	2019-5772
		▮	2019-5805
		▮	2019-5868
Service Workers	Bypass	▮	2018-6093
	Incorrect security	▮	2018-6091
	Insufficient policy	▮	2019-5779
	Others	▮	2019-5823
Shared Web Workers	Insufficient policy	▮	2018-6032
SVG	Type Confusion	▮	2019-5757
third_party_angle	Overflow	■	2018-17466
		▮	2019-5806
		▮	2019-5817
		▮	2019-5836
third_party_boringssl	Disclosure	■	2017-15423
third_party_ffmpeg	Others	■	2017-1000460
third_party_icu	Overflow	■	2017-15422
third_party_libvpx	UAF	▮	2018-6155
		▮	2019-5764
third_party_libxml	Overflow	▮	2017-5130
	UAF	▮	2017-15412
third_party_sqlite	OOB read	▮	2019-5827
Timing	Disclosure	▮	2018-6134
URL	Incorrect security	▮	2019-5839
		▮	2018-6042
Video, Web Audio	Bypass	▮	2018-6168
	Disclosure	▮	2018-6177
Web Assembly	Incorrect security	▮	2018-6116
		▮	2018-6131
	Insufficient validation	▮	2018-6036
	OOB write	▮	2017-15401
	Others	■	2017-5132
		▮	2018-6061
	Overflow	l	2018-6092
	UAF	▮	2017-15399
Web Audio	Bypass	▮	2018-6161
	Others	■	2018-16067
	UAF	■	2018-18339
		■	2018-6060
		▮	2017-5129
Web Bluetooth	Spoof	■	2018-16079
Web MIDI	UAF	▮	2019-5789
Web SQL	Overflow	■	2018-20346
WebGL	Insufficient policy	▮	2018-6047
	Insufficient validation	■	2018-6034
	Overflow	■	2017-5128
		▮	2018-6038
		▮	2018-6162
WebGPU		▮	2018-17470
		▮	2018-17470
		▮	2018-6073
		▮	2018-6154
		▮	2019-5770
WebRTC	Incorrect security	▮	2018-6130
	OOB read	▮	2018-16083
		▮	2018-6129
	Others	▮	2018-6132
	Overflow	▮	2018-6156
	Type Confusion	▮	2018-6157
	UAF	■	2019-5760
		▮	2018-16071
XMLHttpRequest	Incorrect security	■	2019-5832

REFERENCES

[1] G. J. Holzmann, "Code Inflation," *IEEE Software*, vol. 32, no. 2, Mar. 2015.

[2] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[3] *The Heartbleed Bug*, http://heartbleed.com/.

[4] *CVE-2014-0038: Privilege Escalation in X32 ABI*, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038, 2014.

[5] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective Program Debloating via Reinforcement Learning," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[6] Wikipedia, *Hamming distance*, https://en.wikipedia.org/wiki/Hamming_distance, 2020.

[7] Chromium, *Chrome platform status*, https://chromestatus.com/features.

[8] J. Martellaro, *Why Your iPhone Uses PNG for Screen Shots and JPG for Photos*, https://www.macobserver.com/tmo/article/why-your-iphone-uses-png-for-screen-shots-and-jpg-for-photos.

[9] ImageTragick, *ImageMagick Is On Fire: CVE-2016-3714*, https://imagetragick.com/.

[10] C. Mulliner and M. Neugschwandtner, *Breaking payloads with runtime code stripping and image freezing*, 2015.

[11] T. L. Yurong Chen and G. Venkataramani, "Damgate: Dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[12] A. G. Hashim Sharif Muhammad Abubakar and F. Zaffar, "Trimmer: Application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[13] S. Mishra and M. Polychronakis, "Shredder: Breaking Exploits through API Specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[14] X. Z. Zhongshu Gu Brendan Saltaformaggio and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[15] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schroder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[16] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the os: On the practicality of operating system kernel debloating," in *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[17] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proceedings of the 2013 international symposium on memory management (ISMM)*, 2013.

[18] D. W. Yufei Jiang and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

[19] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019.

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

[21] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[22] H. Xue, Y. Chen, G. Venkataramani, and T. Lan, "Hecate: Automated customization of program and communication features to reduce attack surfaces," in *International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2019.

[23] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[24] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you don't," in *Proceedings of the 41st*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020.

[25] P. Snyder, C. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2017.

[26] snyderp, *Some blocked features still accessible*, https://github.com/snyderp/web-api-manager/issues/97, 2018.

[27] A. Quach and A. Prakash, "Bloat factors and binary specialization," in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2019.

[28] M. D. Brown and S. Pande, "Carve: Practical security-focused software debloating using simple feature set mappings," in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2019.

[29] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically Debloating Containers," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[30] Microsoft, *Application Inspector*, https://github.com/microsoft/ApplicationInspector, 2020.

[31] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*, 2019.

[32] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[33] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[34] Google, *V8 JavaScript Engine*, https://chromium.googlesource.com/v8/v8.git.

[35] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao, "Software Watermarking Using Return-Oriented Programming," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[36] D. Bruening and S. Amarasinghe, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[38] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[39] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 325384-068US. Nov. 2018, vol. 3 (3A, 3B, 3C & 3D): System Programming Guide, ch. 35.

[40] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security*, 2018.

[41] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[42] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, 2017.

[43] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[44] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[45] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing Functions in Binaries with Neural Networks," in *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[46] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[47] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[48] *Function and Macro Index*, https://www.gnu.org/software/libc/manual/html_node/Function-Index.html.

[49] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[50] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[51] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[52] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, 2017.

[53] *The Top 500 Sites on the Web*, https://www.alexa.com/topsites.

[54] *Octane*, https://chromium.github.io/octane.

[55] *SunSpider*, https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.

[56] *Dromaeo-JS*, http://dromaeo.com/?dromaeo.

[57] *Dromaeo-DOM*, http://dromaeo.com/?dom.

[58] T. C. Projects, *Site Isolation*, https://www.chromium.org/Home/chromium-security/site-isolation.

[59] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, "The Web/Local Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[60] P. Snyder, C. Taylor, and C. Kanich, "Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[61] T. C. Projects, *Getting Around the Chromium Source Code Directory Structure*, https://www.chromium.org/developers/how-tos/getting-around-the-chrome-source-code, 2020.

[62] W3C, *All standars and drafts*, https://www.w3.org/TR/, 2020.

[63] Caniuse.com, *Support tables for HTML5, CSS3, etc.* https://caniuse.com/#feat=feature-policy, 2020.

[64] Alexa, *The top 500 sites on the web*, https://www.alexa.com/topsites, 2020.

[65] P. Papadopoulos, P. Ilia, M. Polychronakis, E. P. Markatos, S. Ioannidis, and G. Vasiliadis, "Master of web puppets: Abusing web browsers for persistent and stealthy computation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[66] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining elf static linking, semantically," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[67] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[68] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[69] *Intel Processor Trace Tools*, https://software.intel.com/en-us/node/721535, Accessed: 2018-11-1.

[70] *american fuzzy lop*, http://lcamtuf.coredump.cx/afl/, Accessed: 2020-2-12, 2020.

[71] Electron, https://www.electronjs.org/, 2020.

[72] C. Details, *Vulnerabilities statistics on Google Chrome*, https://www.cvedetails.com/product/15031/Google-Chrome.html?vendor_id=1224, 2019.

[73] N. I. of Standards and Technology, *National Vulnerability Database*, https://nvd.nist.gov/, 2020.

[74] Chromium, *Blink (rendering engine)*, https://www.chromium.org/blink.

[75] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos, "Shrinkwrap: Vtable protection without loose ends," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015.

[76] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing C++ virtual calls from memory corruption attacks," in *21st Annual Network and Distributed System Security Symposium*, 2014.

[77] A. Prakash, X. Hu, and H. Yin, "Vfguard: Strict protection for virtual function calls in cots c++ binaries," 2015.

[78] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc and llvm," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[79] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song, "Vtrust: Regaining trust on virtual calls.," in *NDSS*, 2016.

[80] LLVM, *The often misunderstood gep instruction*, https://llvm.org/docs/GetElementPtr.html.

[81] L. O. Andersen, "Program analysis and specialization for the c programming language," Tech. Rep., 1994.

[82] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[83] Wikipedia, *Component (graph theory)*, https://en.wikipedia.org/wiki/Component_(graph_theory).

[84] ——, *Strongly connected component*, https://en.wikipedia.org/wiki/Strongly_connected_component.