RAZOR: A Framework for Post-deployment Software Debloating

Chenxiong Qian*, Hong Hu*, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, Wenke Lee

Georgia Institute of Technology

Abstract

Commodity software typically includes a large number of functionalities for a broad user population. However, each individual user usually only needs a small subset of all supported functionalities. The bloated code not only hinders optimal execution, but also leads to a larger attack surface. Recent works have explored program debloating as an emerging solution to this problem. Unfortunately, these works require program source code, limiting their real-world deployability.

In this paper, we propose a practical debloating framework, RAZOR, that performs code reduction for deployed binaries. Based on users’ specifications, our tool customizes the binary to generate a functional program with minimal code size. Instead of only supporting given test cases, RAZOR takes several control-flow heuristics to infer complementary code that is necessary to support user-expected functionalities. We evaluated RAZOR on commonly used benchmarks and real-world applications, including the web browser FireFox and the close-sourced PDF reader FoxitReader. The result shows that RAZOR is able to reduce over 70% of the code from the bloated binary. It produces functional programs and does not introduce any security issues. RAZOR is thus a practical framework for debloating real-world programs.

1 Introduction

“Entities are not to be multiplied without necessity.”
— Occam’s Razor

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 [16, 39]. 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 up unnecessary attack vectors. Indeed, many serious vulnerabilities are rooted in the features that most users never use [31, 35]. Therefore, security researchers are beginning to explore software debloating as an emerging solution to this problem.

Unfortunately, most initial works on software debloating rely on the availability of program source code [40, 15, 44], which is problematic in real-world use. First, most users do not have access to the source code, and even if they do, it is challenging for them to rebuild the software, diminishing the intended benefits of software bloating. Moreover, users may use the same software in drastically different ways, and thus the unnecessary features to be removed will accordingly vary from user to user. Therefore, to obtain the most benefits, the debloating process should take place after software deployment and should be tailored for each individual user.

Making such a post-deployment approach beneficial and usable to end-users creates two challenges: 1) how to allow end-users, who have little knowledge of software internals, to express which features are needed and which should be removed and 2) how to modify the software binary to remove the unnecessary features while keeping the needed ones.

To address the first challenge, we can ask end-users to provide a set of sample inputs to demonstrate how they will use the software, as in the CHISEL work [15]. Unfortunately, programs debloated by this approach only support given inputs, presenting a rather unusable notion of debloating: if the debloated software only needs to support an apriori, fixed set of inputs, the debloating process is as simple as synthesizing a map from the input to the observed output. However, from our experiments, we find that even processing the same input multiple times will result in different execution paths (due to some randomization factors). Therefore, the naive approach will not work even under simplistic scenarios.

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. Furthermore, if the user provides some description of all possible inputs (e.g., patterns), it is still hard to identify all reachable code for those inputs. Thus, we be-
lieve that any debloating mechanism in the post-deployment setting will be based on best-effort heuristics. The heuristics should help identify the related-code as much as possible, and meanwhile include minimal functionally unrelated code. Note that techniques like dead code elimination [23, 22] and delta debugging [49, 42] do not apply to this problem because they only focus on either removing static dead code or preserving the program’s behavior on a few specific inputs.

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 small increase of the final code size.

Once all the related-code is identified, we develop a binary-rewriting platform to remove unnecessary code and generate a debloated program. Thanks to the nature of program debloating, our platform does not face the symbolization problem from general binary-rewriting tools [51, 53, 52, 5]. Specifically, a general binary-rewriting tool has to preserve all program functionalities, which is difficult without a reliable disassembling technique and a complete control-flow graph (CFG) [2]. For debloating, we preserve only the functionalities related to the sample inputs, where the disassembling and CFG are available by observing the program execution.

We designed the RAZOR framework to realize the post-deployment debloating. The framework contains three components: Tracer monitors the program execution with the given sample inputs to record all executed code; PathFinder utilizes our heuristics to infer more related-code from the executed ones; Generator generates a new binary based on the output of Tracer and PathFinder. In the RAZOR framework, we implemented three tracers (two based on dynamic binary instrumentation and one based on a hardware tracing feature), four path finding heuristics, and one binary generator.

To understand the efficacy of RAZOR on post-deployment debloating, 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 different 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.

In summary, we make the following contributions:

- **New approach.** We proposed a practical post-deployment debloating framework that works on program binaries. Besides given test inputs, our system supports more inputs of the required functionalities.

- **Open source.** We designed RAZOR as an end-to-end system to produce a minimal functional executable. We implemented our system on an x86-64 Linux system and will open source RAZOR at https://github.com/cxreet/razor.

- **Practical and ready-to-use.** We evaluated RAZOR on real-world programs such as Firefox and FoxitReader and showed that these programs can be significantly debloated, resulting in better security.

## 2 Problem

### 2.1 Motivating Example

Figure 1a 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 `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 1a 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 [27]. In this case, the code is bloated with the unnecessary JPEG parser, which may contain security bugs [18]. 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.
In this paper, we develop techniques to remove user-specified unnecessary functionalities from bloated programs. Given a program \( P \) that has a set of functionalities \( \mathcal{F} = \{ F_0, F_1, F_2, \ldots \} \) and a user specification of necessary functionalities \( \mathcal{F}_u = \{ F_i, F_j, F_k, \ldots \} \), our goal is to generate a new program \( P' \) that only retains functionalities in \( \mathcal{F}_u \) and gracefully refuses requests of other functionalities in \( \mathcal{F} - \mathcal{F}_u \).

The program in Figure 1a has two high-level functionalities: parsing PNG images and parsing JPEG images, while the user specification only requires the first functionality. In this case, the goal of debloating is to generate minimal code that only supports parsing PNG images while exiting gracefully if the given images are in other formats. From the simple code we can easily tell that code in the yellow background (i.e., line 6, 7 and 15) is not necessary, so we remove such code in a safe manner: function parseJPEG will be simply removed; for line 6 and 7, we should replace the code with fault-handling code to prompt warnings and exit gracefully.

In this paper, we focus on reducing functionalities from software binaries. Specifically, the program \( P \) is given as a binary, while the source code like Figure 1a is not available. Instead, we construct the control-flow graph (CFG) from the executable and use it to guide the binary debloating. Figure 1b and Figure 1c show CFGs of the bloated binary and the debloated one, respectively. Black arrows represent intra-procedural jumps, while dotted arrows stand for inter-procedural calls and returns. Originally, function imageParser can execute lines 6 and 7 and invoke function parseJPEG. In the debloated binary, these lines and functions are not reachable, and the CFG is simplified to Figure 1c. For the vulnerability in the removed code, the new binary prevents attackers from triggering them in the first place.

### 2.2 Program Debloating

In this paper, we develop techniques to remove user-specified unnecessary functionalities from bloated programs. Given a program \( P \) that has a set of functionalities \( \mathcal{F} = \{ F_0, F_1, F_2, \ldots \} \) and a user specification of necessary functionalities \( \mathcal{F}_u = \{ F_i, F_j, F_k, \ldots \} \), our goal is to generate a new program \( P' \) that only retains functionalities in \( \mathcal{F}_u \) and gracefully refuses requests of other functionalities in \( \mathcal{F} - \mathcal{F}_u \).

The program in Figure 1a has two high-level functionalities: parsing PNG images and parsing JPEG images, while the user specification only requires the first functionality. In this case, the goal of debloating is to generate minimal code that only supports parsing PNG images while exiting gracefully if the given images are in other formats. From the simple code we can easily tell that code in the yellow background (i.e., line 6, 7 and 15) is not necessary, so we remove such code in a safe manner: function parseJPEG will be simply removed; for line 6 and 7, we should replace the code with fault-handling code to prompt warnings and exit gracefully.

In this paper, we focus on reducing functionalities from software binaries. Specifically, the program \( P \) is given as a binary, while the source code like Figure 1a is not available. Instead, we construct the control-flow graph (CFG) from the executable and use it to guide the binary debloating. Figure 1b and Figure 1c show CFGs of the bloated binary and the debloated one, respectively. Black arrows represent intra-procedural jumps, while dotted arrows stand for inter-procedural calls and returns. Originally, function imageParser can execute lines 6 and 7 and invoke function parseJPEG. In the debloated binary, these lines and functions are not reachable, and the CFG is simplified to Figure 1c. For the vulnerability in the removed code, the new binary prevents attackers from triggering them in the first place.

### 2.3 Challenges and Solutions

From the previous example, we can find the gap between the user specification and the code removed: users specify that the functionality of parsing PNG files is necessary (i.e., others are unnecessary), while we finally remove line 6, line 7, and function parseJPEG. However, mapping high-level functionalities to low-level code manually is challenging, especially for large programs. Specifically, this leads to two general challenges of program debloating:

**C1.** How to express unnecessary functionalities;

**C2.** How to map functionalities to program code.

One possible solution is to rely on end-users to provide a set of test cases for each necessary/unnecessary functionality so we can inspect the program execution to learn the related program code. Our problem can be rephrased as follows: given the program binary \( P_b \) and a set of test cases \( T = \{ t_i, t_j, t_k, \ldots \} \), where each test case \( t_i \) triggers some functionalities of \( P_b \), we will create a minimal program \( P'_b \) that supports and only supports functionalities triggered by the test cases in \( T \).

Test cases help us address challenges **C1** and **C2**. 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 1a 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. Our evaluation in §5.2 shows that simply removing all non-executed code introduces many bugs, even exploitable ones, to the debloated program. Therefore, a test-case-based debloating system faces the following challenge.

**C3.** How to find more related-code from limited test cases.

To address challenge **C3**, we propose control-flow-based heuristics to infer more related-code that is necessary to support the required functionalities but was missed during our
inspection. Suppose the test cases in $T$ only trigger the execution of instructions in $\mathcal{S} = \{i_0, i_1, i_2, \ldots\}$, our heuristic will automatically infer more code that is related to the functionalities covered by $T$. Specifically, we identify a super set $\mathcal{S}' = \mathcal{S} \cup \{i_3, i_4, i_5, \ldots\}$ and keep all instructions in $\mathcal{S}'$ while removing others to minimize the code size. When debloating the code in Figure 1a, the execution of given test cases does not cover line 12. However, with our heuristics, we will include this line in the debloated program. The evaluation in §5.3 shows that our heuristic is effective in finding related-code paths and introduces only a small increase in code size.

3 System Design

Figure 2 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 (§3.1). 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 (§3.2). The expanded CFG contains non-executed instructions that are necessary for completing the required functionalities. In the end, with the expanded CFG, Generator rewrites the original binary to produce a minimal version that only supports required functionalities (§3.3).

3.1 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 [13]), or overlapped basic blocks (e.g., legitimate code reuse [26]), 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 3 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 [7, 25, 6]. Hardware-based logging can record efficiently [20] but requires particular hardware and may not guarantee the completeness (e.g., data loss in Intel PT [17]). Second, program executions under different tracing environ-
ments will show divergent paths. For example, Dynamo 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 (details in §4.1) 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 [52, 51, 3, 4, 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 [47]. 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 [53]. For example, `jmpq *0x4e65a0(%rdx,8)` indicates a jump table starting from address `0x4e65a0`. By identifying more instructions, we are able to include them in the binary if our heuristic treats them as related-code.

### 3.2 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 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 \( \log(\sqrt{\text{abs}(\max(rax,rbx,rcx)))} \). Dashed branches and blocks are not executed during tracing, while others are executed. The original execution path is \( L1 \rightarrow L2 \rightarrow L3 \rightarrow L5 \rightarrow L7 \rightarrow L9 \). Blocks \( L4, L6, L8 \), and the branch \( L1 \rightarrow L3 \) are missed in the original CFG. With the \( \text{zCode} \) heuristic, PathFinder adds branch \( L1 \rightarrow L3 \) 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 [53, 3] to perform binary analysis. When applying the zCall heuristic on the example in Figure 4, PathFinder further includes block \( L4 \), and path \( L3 \rightarrow L4 \rightarrow L5 \), 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. However, zLib does not allow calls to non-executed external functions. In Figure 4, with this heuristic, PathFinder adds block \( L6 \) and path \( L5 \rightarrow L6 \rightarrow L7 \) 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
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 */
for cnd_br ∈ CFG:
  nbb = get_non_taken_branch(cnd_br)
  if nbb = NULL: continue
  if heuristic >= zCode and nbb ∈ CFG:
    CFG' = CFG' ∪ {cnd_br⇒nbb}
  paths = get_alternative_paths(CFG', nbb)
  for p ∈ paths:
    include = false
    if heuristic = = zCall: include = !has_call(p)
    elif heuristic = = zLib: include = !has_new_libcall(p)
    elif heuristic = = zFunc:
      include = !has_new_func(CFG', p, libcall_groups)
    if include:
      CFG' = CFG' ∪ p

descriptions and group them manually. For libc functions, we classify the ones that fall into the same subsection in [32] 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).

3.3 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 [11]). We discuss this design choice in §6.

3.3.1 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 of how we manipulate the assembly file. Figure 5 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 0x6 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 §3.3.3). 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.
3.3.2 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 [52, 51]. 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 [53]. 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 [53], 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 5 gives an example of indirect call concretization. In the execution trace, instruction call %rax in block 0x70 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 [53] 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 [12, 43, 9]. 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-frequency targets first. The targets inferred from heuristics have a frequency of zero. With this optimization, we can reduce the overhead significantly.

3.3.3 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 5 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 Implementation

We implement a prototype of RAZOR with 1,085 lines of C code, 514 lines of C++ code, and 4,034 lines of python code, as shown in Table 1. The prototype currently supports x86-64 ELF binaries. Our design is platform-agnostic and we plan to support other binary formats from different architectures. We tried our system on system libraries (e.g., libc.so, libm.so) and report our findings in §6.

4.1 Tracer Implementations

As we discussed in §3.1, each tracing method has different benefits and limitations, such as the tracing efficiency and completeness. We provide three different implementations of
we do not know the complete set of targets. Considering the overhead Tracing with hardware feature.

We use the dy-
Tracing with software instrumentation.

We use RAZOR to de-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.

![Figure 6: Code size reduction on two benchmarks.](image)

<table>
<thead>
<tr>
<th>Component</th>
<th>Tracer</th>
<th>PathFinder</th>
<th>Generator</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1,085</td>
<td>0</td>
<td>0</td>
<td>1,085</td>
</tr>
<tr>
<td>C++</td>
<td>514</td>
<td>0</td>
<td>0</td>
<td>514</td>
</tr>
<tr>
<td>Python</td>
<td>218</td>
<td>743</td>
<td>3,073</td>
<td>4,034</td>
</tr>
</tbody>
</table>

Table 1: Implementation of different RAZOR components.

Tracer in RAZOR so that users can choose the best one for their purpose. In our evaluation, we use software-based instru-
mention to collect complete traces for simple programs, and use a hardware-based method to efficiently get trace from large programs.

**Tracing with software instrumentation.** We use the dy-
mamic instrumentation tools Dynamorio [7] and Pin [25] to monitor the execution of the bloated program. Both tools provide instrumentation interfaces at function level, basic block level, and instruction level. We implement three instrument-
ation passes to collect control-flow information. First, at the beginning of each basic block we record its start address; second, for each conditional jump instruction, we insert two pieces of code between the instruction and its two targets to log the taken information; third, before each indirect call and jump instruction, we record the concrete target for each invocation. At runtime, we remove the basic block instruction immediately after its first execution to avoid unnecessary overhead. Similarly, we remove the instrumentation of conditional branches once that branch has been taken. However, we keep the instrumentation of indirect call and jump instructions, as we do not know the complete set of targets.

**Tracing with hardware feature.** Considering the overhead of software instrumentation, we provide an efficient Tracer built on Intel Processor Trace (Intel PT) [20]. Intel PT records the change of flow information in a highly compressed manner: the TNT packet describes whether one conditional branch is taken or non-taken; the TIP packet records the target of indirect branches, like indirect call and return. As Intel PT directly writes the trace to physical memory without touching the page table or memory cache, it achieves the most efficient tracing. Our Tracer decodes the traces from Intel PT to get necessary control-flow information. We can use other hardware features available on different platforms to implement efficient Tracer, like branch trace store (BTS)
on Intel CPUs or program flow trace (PTM) on ARM CPUs.

**4.2 Update ELF Exception Handler**

ELF binaries generated by gcc and clang adopt the table-based exception handling [46] to provide stack unwind and exception handler information. Specifically, ELF keeps a table in the .eh_frame_hdr section, one entry per function. Each entry indicates the location of a frame description entry (FDE) in the .eh_frame section, which further specifies the location of the language-specific data area (LSDA). The LSDA region in the .gcc_except_table section contains the concrete address of exception handlers, called landingpad.

We have to replace the old value of all landingpads in .gcc_except_table with the new ones. However, the challenge is that the value in .gcc_except_table is encoded in the LEB128 format – a variable-length encoding that may have different lengths for different values. Since we update the old address with a different one, the encoding of the new address may take more bytes and thus cannot be put into the original location. To solve this problem, we update the section layout of the binary to create more space for the new address. Specifically, we shrink the table inside the .eh_frame_hdr section to exclude entries of non-executed functions. Recall that the given test cases only trigger part of the functionalities, and the non-executed functions will not be included in the debloated binary. Then we shift .eh_frame and .gcc_except_table sections to get more space for our update of landingpad values.

**5 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? (§5.1)
- **Functionality.** Does the de-bloated binary support the functionalities in given test cases? (§5.2) How effective is PathFinder in finding complementary code? (§5.3)
- **Security.** Does RAZOR reduce the attack surface of the debloated binaries? (§5.4)
100%  RAZOR  Chisel  HISEL  AVERAGE 100%  80%  60%  40%  20%  0%  (a) basic blocks  100%  80%  60%  40%  20%  0%  RAZOR  Chisel  HISEL  AVERAGE  (b) instructions  100%  80%  60%  40%  20%  0%  RAZOR  Chisel  HISEL  AVERAGE  (c) ROP gadgets

Figure 7: Reduction of basic blocks, instructions, and ROP gadgets, debloated by RAZOR and CHISEL from CHISEL benchmarks.

- **Performance.** How much overhead does RAZOR introduce into the debloated binary? (§5.5)
- **Practicality.** Does RAZOR work on commonly used software in the real world? (§5.6)

**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 paper1 [15]; the web browser Firefox 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.

### 5.1 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 6a shows the code reduction of SPEC benchmarks debloated by RAZOR. Figure 6b 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, hammer, gamess, and tar, RAZOR removes more than 90% of the original code. For bwaves, zeusmp, and Gem5FDTD, 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 (§5.2) and security issues (§5.4).

**Other reduction metrics.** We also measured RAZOR’s effectiveness on reducing basic blocks (Figure 7a) and instructions (Figure 7b) from CHISEL benchmarks and compared these results with those achieved by CHISEL. On average, RAZOR removes 53.1% of basic blocks and 63.3% of instructions from the original programs, while CHISEL reduces 66.0% of basic blocks and 88.5% of instructions from the same set of programs. This result is consistent with the code size reduction, where RAZOR reduces less code, as it can neither remove any executed-but-unnecessary blocks or instructions, nor utilize compiler to aggressively optimize the debloated code.

### 5.2 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 2 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,

---

1We appreciate the help of CHISEL authors for sharing the source code and their benchmarks.
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 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 strftime 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 though it successfully finds matched strings. We find that in the verification process of CHISEL, all output of the debloated binaries is redirected to the /dev/null device. Therefore, it cannot detect any missing or inconsistent output.

### Table 2: Failed test cases by RAZOR binaries and CHISEL binaries.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th># of Tests</th>
<th>Failed by Chisel W</th>
<th>I</th>
<th>C</th>
<th>M</th>
<th>Failed by RAZOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>1.0.5</td>
<td>6</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>chown</td>
<td>8.2</td>
<td></td>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>date</td>
<td>8.21</td>
<td>50</td>
<td>5</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>grep</td>
<td>2.19</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>gzip</td>
<td>1.2.4</td>
<td>5</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>mkdir</td>
<td>5.2.1</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>rm</td>
<td>8.4</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>sort</td>
<td>8.16</td>
<td>112</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>tar</td>
<td>1.14</td>
<td>26</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>4</td>
<td>- (zLib)</td>
</tr>
<tr>
<td>uniq</td>
<td>8.16</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>- (zLib)</td>
</tr>
</tbody>
</table>

Figure 8 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.

### 5.3 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 §5.6.1.

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 training 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 7 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 8 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 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 9 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 11 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 12 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 10 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 `strncpy`, the debloated binary fails even with the zLib heuristic. However, the training inputs ever invoke function `strncpy`, which has the functionality similar to `strncpy` (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.

### 5.4 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 vulnera-
bilities, including the ones shown in the current version and the ones only in earlier versions. For 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 3 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 [15] (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 bzex, not by the bzip2 binary. Therefore, RAZOR will not disable such bugs.

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 §5.2, where CHISEL achieves higher code reduction but fails some expected functionalities.

Reduction of ROP gadgets. We also measured the reduction of ROP gadgets. Once the attacker is able to divert the control-flow, the number of reusable ROP gadgets indicate the vulnerability of the program to control-flow hijacking attacks. Figure 7c show that RAZOR reduces 61.9% ROP gadgets, while CHISEL reduces 85.1% ROP gadgets. Although RAZOR achieves less ROP gadget reduction, this result is expected. In the design of RAZOR, we intentionally pay more attention on preventing forward-edge control-flow attacks, where attackers corrupt function pointers, instead of return addresses, to diver the control-flow. As shadow stack technique are getting deployed in compilers [24] and even hardware [19], our technique of indirect call/jump concretization (§3.3.2) complements existing practical return-protections to achieve complete control-flow integrity.

5.5 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 13. 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 execution 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.

5.6 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 [28]. 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 shows the code reduction rate and the number of failed cases of debloated binaries with different path-finding heuristics. Both Firefox
websites can benefit with at least 5.0% code reduction for
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 [38, 21, 48]. 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.
Table 4 shows the debloating result, the
used path-finding heuristic and the security benefits over the
general debloating in Table 4. 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.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Firefox crash-sites reduction</th>
<th>FoxitReader crash-PDFs reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>13 67.6%</td>
<td>39 89.8%</td>
</tr>
<tr>
<td>zCode</td>
<td>13 68.0%</td>
<td>39 89.9%</td>
</tr>
<tr>
<td>zCall</td>
<td>2 63.1%</td>
<td>5 89.4%</td>
</tr>
<tr>
<td>zLib</td>
<td>0 60.1%</td>
<td>0 87.0%</td>
</tr>
<tr>
<td>zFunc</td>
<td>0 60.0%</td>
<td>0 87.0%</td>
</tr>
</tbody>
</table>

Table 4: Debloating Firefox and FoxitReader with RAZOR, together with different path-finding heuristics.

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
[33], SunSpider [34], Dromaeo-JS [30], and Dromaeo-DOM
[29] 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 [38, 21, 48]. 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 and security-sensitive websites: banking websites, websites for electronic commerce, and social media websites. Table 6 shows the debloating result, the
used path-finding heuristic and the security benefits over the
general debloating in Table 4. 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.

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Firefox crash-sites reduction</th>
<th>FoxitReader crash-PDFs reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>13 67.6%</td>
<td>39 89.8%</td>
</tr>
<tr>
<td>zCode</td>
<td>13 68.0%</td>
<td>39 89.9%</td>
</tr>
<tr>
<td>zCall</td>
<td>2 63.1%</td>
<td>5 89.4%</td>
</tr>
<tr>
<td>zLib</td>
<td>0 60.1%</td>
<td>0 87.0%</td>
</tr>
<tr>
<td>zFunc</td>
<td>0 60.0%</td>
<td>0 87.0%</td>
</tr>
</tbody>
</table>

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
[33], SunSpider [34], Dromaeo-JS [30], and Dromaeo-DOM
[29] 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.

5.6.1 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 5. 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.

<table>
<thead>
<tr>
<th>Train/Test ID</th>
<th>#Failed</th>
<th>Reduction</th>
<th>failed websites</th>
</tr>
</thead>
<tbody>
<tr>
<td>T00</td>
<td>0</td>
<td>59.3%</td>
<td>wordpress.com</td>
</tr>
<tr>
<td>T01</td>
<td>2</td>
<td>59.1%</td>
<td>wordpress.com, twitch.tv</td>
</tr>
<tr>
<td>T02</td>
<td>2</td>
<td>59.3%</td>
<td>wordpress.com, twitch.tv</td>
</tr>
<tr>
<td>T03</td>
<td>2</td>
<td>59.1%</td>
<td>wordpress.com, twitch.tv</td>
</tr>
<tr>
<td>T04</td>
<td>0</td>
<td>59.2%</td>
<td></td>
</tr>
<tr>
<td>T05</td>
<td>1</td>
<td>59.1%</td>
<td>aliexpress.com</td>
</tr>
<tr>
<td>T06</td>
<td>0</td>
<td>59.2%</td>
<td></td>
</tr>
<tr>
<td>T07</td>
<td>0</td>
<td>59.1%</td>
<td></td>
</tr>
<tr>
<td>T08</td>
<td>2</td>
<td>59.3%</td>
<td>wordpress.com, twitch.tv</td>
</tr>
<tr>
<td>T09</td>
<td>0</td>
<td>59.1%</td>
<td></td>
</tr>
</tbody>
</table>

Table 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.

5.6.1 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 5. 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.
was removed. We find that code related to features such as record/replay, integer/string conversion, compression/decompression are removed.

6 Discussions

Best-effort path inference. Mapping high-level functionalities to low-level code is known to be challenging, especially when source code is unavailable. RAZOR empirically adopts control-flow-based heuristics to infer more related-code with its best effort. We understand that such a heuristic cannot guarantee the completeness or soundness of the path inference, and the debloated binary may miss necessary code (i.e., code for handling different environment variables) or include unnecessary ones (like some initialization code). However, we noticed that the heuristic-based method has been widely used in binary analysis and rewriting [53, 52]. With the execution trace, RAZOR is able to mitigate some limitations of these works, such as finding indirect call targets. Further, the evaluation result demonstrates that our control-flow-based heuristics are practically effective.

CFI and debloating. Control-flow integrity (CFI) enforces that each indirect control-flow transfer (i.e., indirect call/jump and return) goes to legitimate targets [1]. It prevents malicious behaviors that are unexpected by program developers. In contrast, software debloating removes benign-but-unnecessary code based on users’ requirements. For example, if function A is designed to be a legitimate target of an indirect call i, CFI will allow the transfer from i to A. However, if the user does not need the functionality in A, software debloating will disable the transfer and completely remove the function code. In fact, CFI and debloating are complementary to each other. On the one hand, debloating achieves a coarse-grained CFI where an attacker can only divert the control-flow to remaining code. It also simplifies the analysis required by some CFI works [50, 37] because of a smaller code base. On the other hand, existing CFI works provide fundamental platforms for enforcing debloating. For example, RAZOR makes use of several binary analysis techniques developed in binCFI [53] for optimization.

Library debloating. We tried to use RAZOR to debloat system libraries for each program. Our tool works well on some libraries (e.g., 1ibm.so and 1ibgcc.so), but fails on others. For example, the debloated 1ibc.so triggers a different execution path even if we aggressively include more related-code with the zFunc heuristic. After inspecting the failure cases on 1ibc.so, we found that its execution path is very sensitive to the change of the execution environment. One reason is that 1ibc.so contains a lot of highly optimized code for memory or string operations (e.g., memcmp), which, based on the argument value, choose the most efficient implementation. For example, function memcmp implements 16 different subroutines to process strings with different alignments. Another reason is that it performs different executions according to the process status. For example, for each memory allocation, malloc searches a set of cached chunks and picks up the first available one. Inputs with different sizes may cause malloc to walk through a complete non-executed path. From such a preliminary result, we plan to develop library-specific heuristics to handle environment-sensitive executions. For example, we can perform debloating on the function level instead of the current basic block level. We also plan to explore existing library debloating solutions that work on source code [40] and port them into binaries if necessary.

Removing original code. The current design of RAZOR keeps the original code section inside the debloated program and changes its permission to read-only to reduce the attack surface. This design simplifies the handling of potential data inside the code section, which the program may read for special purposes. For example, LLVM will emit jump tables in the code section to support efficient switch statements [11], and the indirect jump instruction will obtain its targets by reading the table. To further reduce the program size and memory usage, we can completely remove the original code section as follows: 1) during the execution tracing, we set the original code section to execute-only [11] so that any read from the code section will trigger the exception and can be logged by Tracer; 2) we perform backward data-flow analysis to identify the source of the data pointer used for each logged memory access; 3) during the binary synthesization, we relocate the data from the original code section to a new data section and update the new code to visit the new location. In this way, we are able to handle the challenging problem of data relocation during binary rewriting. In fact, we performed a study to understand the prevalence of these problems and found that for all the programs tested in the paper, none of them ever reads any data from the code section, given the test cases we used. In these cases, we can simply remove the original code section to minimize the file size and memory footprint.

Future work. We will release the source code of RAZOR. We plan to extend the platform to support binaries in more
formats and architectures, including shared libraries, 32-bit binaries, Windows PE programs, MacOS March-O programs, and ARM binaries. At the same time, we will design more security-related heuristics to make RAZOR support various real-world situations.

7 Related Work

Library debloating. Program libraries are designed to support a large number of functionalities for different users. Library debloating customizes the general code base for each program and leads to significant code reduction. Mulliner et al. propose CodeFreeze to remove the unnecessary functionalities from Windows shared libraries [36]. They start from per-library control-flow analysis to identify the code dependency of each exported function. Then they check the program binary to find all required library functions. By stitching program required functions and per-library CFG, they rewrite the library to remove unreachable code region. Similarly, Quach et al. [40] present library debloating through piece-wise compilation and loading. Instead of customizing the library for each program, they split the large library into small groups based on the control-flow dependency. At runtime, they use a customized loader to rewrite the library code to remove unnecessary functions. Jiang et al. [23, 22] propose to remove dead code from Android Apps, Java Runtime Environment, and SDKs. Our system is different from library debloating in two ways. First, previous work performs the binary rewriting at the beginning of each process, leading to performance overhead for each execution, while RAZOR generates the debloated binary through static binary rewriting, which is only performed once and used forever. Second, library debloating utilizes static analysis to find the unused code and has to conservatively keep all potentially useful code. In contrast, our system relies on a dynamic execution trace to locate the code that is executed during tracing or inferred with our heuristic and removes all others.

Delta debugging. Delta debugging is proposed to minimize bug-triggering inputs. For example, Regehr et al. [42] propose C-Reduce to generate a smaller test cases efficiently. Sun et al. [49] present Perses, which exploits formal syntax to generate smaller and functionally equivalent program in a timely manner. Recently, Heo et al. [15] proposed CHISEL, a framework for practical software debloating on program binaries. It utilizes the compiler optimization to reduce code size. These systems, as well as [42, 49, 15], rely on the complicated analysis of program source code, which is not always available for deployed programs. In contrast, RAZOR only requires program binaries, making it more practical for deployment.

Container Debloating. Containers are becoming more popular, and their code base is bloated. Guo et al. [14] proposed a method to monitor the program execution to identify necessary resources and create a minimal container for the traced program. Rastogi et al. [41] developed Cimplifier, which uses dynamic analysis to collect resource usages for different programs and partitions the original container into a set of smaller ones based on user-defined policies. The resulting containers only have resources to run one or more executable programs. The design of RAZOR is also applicable for debloating containers or other systems. For example, Intel PT supports tracing operating systems.

Hardware Debloating. Nowadays, hardware devices are also bloated. For example, general-purpose processors are overly designed for specific applications, such as implantables, wearables, and IoT devices. Cherupalli et al. propose an approach that automatically removes unused gates from the design of a general-purpose processor to generate a bespoke processor for a specific application [10]. On average, the approach can reduce the area by 62% and the power by 50% from the general processor. Currently, software debloating and hardware debloating are performed separately. An interesting direction is to consider both hardware devices and software programs to find more debloating space.

8 Conclusion

In this paper, we presented RAZOR, a framework for practical software debloating on program binaries. It utilizes a set of test cases and control-flow-based heuristics to collect necessary code to support user-expected functionalities. The debloated binary has a reduced attack surface, improved security guarantee, robust functionality, and efficient execution. Our evaluation shows that RAZOR is a practical framework for debloating real-world programs.

Acknowledgment

We thank the anonymous reviewers, and our shepherd, Michael Bailey, for their helpful feedback. This research was supported in part by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP006, by the ONR under grants N00014-17-1-2895, N00014-15-1-2162 and N00014-18-1-2662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA and ONR.
References


[34] SunSpider. https://webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.


Appendix

A Settings for Evaluating PathFinder

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

Table 7: 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.