Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs

Mohamad Mansouri
EURECOM
France

Jun Xu
University of Utah
USA

Georgios Portokalidis
Stevents Institute of Technology
USA

Abstract

Driven by application diversification and market needs, software systems are integrating new features rapidly. However, this "feature creep" can compromise software security, as more code carries the risk of more vulnerabilities. This paper presents a system for disabling features activated by common input types, using a component called F-DETECTOR to detect feature-associated program control flow branches. The system includes a second component called F-BLOCKER to disable features without disrupting application continuity. It does so by treating unwanted features as unexpected errors and leveraging error virtualization to recover execution, by redirecting it to appropriate existing error handling code. We implemented and evaluated the system on the Linux platform using 145 features from 9 programs, and results show that it can detect and disable all features with few errors, hence, outperforming previous works in terms of vulnerability mitigation through debloating.

CCS Concepts

• **Security and privacy → Software and application security**; **Vulnerability management**.

Keywords

Feature removal, tracing, binary analysis, vulnerability removal

ACM Reference Format:
Mohamad Mansouri, Jun Xu, and Georgios Portokalidis. 2023. Eliminating Vulnerabilities by Disabling Unwanted Functionality in Binary Programs. In *Proceedings of 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS'23)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software is continuously growing in terms of functionality and size. This observation led Microsoft's Nathan Myhrvold to define his *First Law of Software*, stating that "software is a gas" because "it expands to fit the container it is in" [22]. However, many users do not use a significant part of the available functionality [37]. We can view this unused set of features as "bloat," which unnecessarily decreases the security and stability of software. In fact, code size and complexity has been linked to bugs by multiple studies [16, 43] and serious vulnerabilities [39] have been discovered in rarely used features. Eliminating feature bloat improves security because it eliminates unnecessary code that may contain known and unknown vulnerabilities.

Many previous works [5, 7, 10, 14, 18, 27, 30, 34, 38] have focused on greedily removing all unused features. They rely on test cases covering part of required functionality to identify the corresponding code and retain it, while erasing all other code, which may result in the unintentional removal of required code that was not covered by the test cases, such as environment-, configuration-, and error-handling code. However, debloating can also be done by only eliminating specific unwanted features, which has a lower chance of erroneously removing required code. Landsborough et al. [19] propose a strategy that uses execution traces collected with unwanted features to erase related code. Unfortunately, this approach relies on the availability of complete test suites for unwanted features and is limited to small, utilities (e.g., `sha1sum`).

In this paper, we present a system for disabling *unwanted* features in binary applications without the above limitations. The system comprises two major components. F-DETECTOR implements a new method for detecting a key control-flow branch in the application that corresponds to the activation of an unwanted feature. Preventing the traversal of that branch, makes the code implementing the feature unreachable and neutralizes any vulnerabilities contained within. F-DETECTOR operates in a semi-automatic way without relying on the program's source code. Users provide a small set of inputs that activate the unwanted features and then follow our guidelines to minimally mutate the inputs for generating new ones that avoid the unwanted feature. To detect the feature-activating branch, F-DETECTOR uses execution traces from both user-provided and mutation-produced inputs, in combination with information obtained through static analysis of the application's binary. F-BLOCKER models the unwanted feature as an unanticipated fault and borrows concepts from software dependability research to handle it gracefully. Technically, F-BLOCKER uses the output of F-DETECTOR and dynamic information to first select a function that can work as a *rescue point*, i.e., a location where execution can rollback and an error can be raised to activate built-in error handling. F-BLOCKER's run time deploys the rescue point, so once the feature entrance is hit, state is rolled back and a valid error code returned.

This work is not the first to target feature removal, but it brings several unique, widely desired advantages. First, it only requires a few inputs and some basic understanding of target-software features (readily available in manuals) to minimally mutate them, decreasing the burden placed upon and required expertise of users. Second, it only disables a single control flow edge, which is more tractable than finding all the code blocks corresponding to a feature and reduces the potential of side effects to other functionality. Third, it can handle different type of programs and features, including features activated by network requests, graphical user interfaces (GUI), file formats, command-line arguments, etc. Finally, it ensures the survival of the application, which is crucial for server programs and larger client applications where crashes can cause data loss.

We have implemented prototypes of F-DETECTOR and F-BLOCKER, which we evaluated using 145 features from 9 applications. To our

ASIACCS'23, July 10–14, 2023, Melbourne, Australia
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

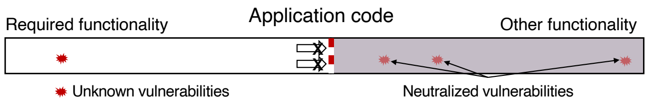


Figure 1: Eliminating vulnerabilities through debloating.

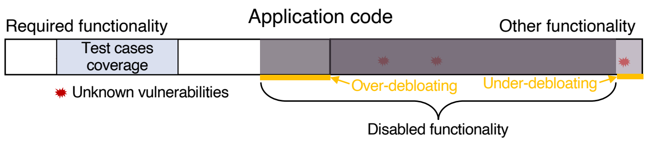


Figure 2: Debloating based on retaining wanted functionality.

knowledge, this is the most extensive experimental evaluation of a system removing unwanted features, where the results are also manually verified. The system was able to detect the correct feature entrance for most of the tested features, regardless of inputs and mutations. Only when handling a small set number of features in BusyBox, did it present errors because we intentionally explored all mutations without strictly following our guidelines. In addition, our system disables 6 known vulnerabilities rooted within the tested features.

In summary, we make the following contributions:

- We define an algorithm for automatically identifying the control-flow edge in a program that dominates a targeted feature, based on dynamically profiling an application with user-selected inputs and static analysis of its binary, which we implement in F-Detector.
- We define a set of guidelines to assist users in selecting the inputs to profile the application.
- We develop an algorithm for automatically defining the self-healing primitives (i.e., rescue points) to disable features while maintaining the continuity of normal service, which we implement in F-Blocker.
- We evaluate the two components using 9 applications and 145 features with 6 associated vulnerabilities (CVEs). Our results show that it can disable all features and insulate the application from the vulnerabilities.
- We compare F-Detector with the current state-of-the-art system, Razor [27], and we find that F-Detector is better at disabling specific features, which leads to the mitigation of more vulnerabilities.

2 Background and Motivation

2.1 Improving Security through Debloating

By disabling undesired functionality during execution, we can neutralize both known and unknown (i.e., zero-day) vulnerabilities, as shown in Fig. 1. Ideally, one can disable unwanted features during compilation before installation. For example, Debian GNU/Linux offers various versions of the popular Vim editor, with *vim-tiny* including only about 12 features compared to the full version's 100+. To handle applications that lack such options, recent research has proposed automatically disabling (i.e., debloating) software by either eliminating all but required functionality or disabling specific unwanted features.

2.2 Different Debloating Strategies

Retaining Wanted Functionality Research [5, 7, 10, 14, 18, 27, 30, 34, 38] in this direction involves the profiling of applications by utilizing test cases to discern essential functionality from non-essential one and erasing or disabling the latter. Their main advantage is the removal of significant code and vulnerabilities.

Disabling Unwanted Functionality Several alternative approaches focus on specific undesired functionality. Some rely on users identifying functions crucial to the desired feature and then employing dynamic and static analysis to eliminate the associated code [15] or blocking its execution [4]. Another technique relies on runtime profiling to identify the code activated when running with unwanted-feature inputs and then overwriting it with no-op (`nop`) instructions [19].

Table 1: Evaluating Razor with Coreutils. ● indicates we discovered a problem and ○ that we did not.

	`bzip2-1.0.5`	`chown-8.2`	`date-8.21`	`grep-2.19`	`gzip-1.2.4`	`mkdir-5.2.1`	`rm-8.4`	`sort-8.16`	`tar-1.14`	`uniq-8.16`
Over-debloating	○	●	○	○	○	○	●	○	●	○
Under-debloating	○	○	●	●	○	●	●	○	○	●

2.3 Retaining vs. Disabling Functionality

Two problems affect debloated applications:

Over-debloating – This issue arises when code associated with necessary functionality is mistakenly disabled. Consequently, the application fails to operate correctly when debloated code paths are activated.

Under-debloating – This problem occurs when some unwanted functionality remains in the program, creating a false sense of security. For example, users may assume that a newly discovered vulnerability does not affect them because the corresponding feature has been disabled. However, some code, including the vulnerable code, remains in the application.

For instance, a recent study [27] has found that Chisel [10], a debloating system based on reinforcement learning, is susceptible to over-debloating. In extreme cases, the system may even remove necessary checks, introducing new vulnerabilities. The same study introduced Razor [27], a system which instead uses a set of heuristics to determine all the code associated with wanted functionality. However, our experiments with the prototype made publicly available by the authors confirm that Razor may also lead to both over- and under-debloating. Table 1 summarizes our findings, where we use the original paper's train and test inputs to detect over-debloating and new test inputs that correspond to unwanted features to identify under-debloating.

The problems of debloating based on wanted functionality are rooted in the fundamental challenge of obtaining perfect inputs that cover all the code required by desired functionality. As a result, trying to remove large amounts of code leads to errors like over-debloating in practice. In contrast, disabling unwanted functionality only requires minimal code trimming and it intuitively reduces the chance of over-debloating. Additionally, recent research [4, 15] suggests that disabling a small but crucial piece of code for the execution of a feature can avoid under-debloating. Therefore, we postulate that disabling unwanted functionality is a more principled approach for avoiding both these problems.

2.4 Limitations of Existing Solutions

Current solutions [4, 15] to disabling unwanted functionality have two main limitations. Firstly, they require manual annotation or comprehension of the implementation to detect code constructs associated with features, making them difficult or even impossible to apply on binaries, and too intricate for non-developers. Secondly, they typically stop unwanted functionality by simply terminating execution (e.g., by replacing unwanted code with an invalid instruction [19]), which is not suitable for servers or applications where data loss could occur when an unwanted feature is used (e.g., image editing applications). Our paper aims to address these limitations and provide a novel solution for removing unwanted functionality.

3 Design Overview

3.1 Key Insight and High-level Idea

Our primary observation is that program functionality, given that it is not always executed independently of the input, is frequently initiated or regulated by a *control-flow branch. Conditional branches* are often utilized to establish state variables that control some functionality. In the example depicted in Fig.3, the branch from line 8 to 9 is only executed for HTTP requests utilizing the PUT method. The request's state update on line 9, later leads to the execution of PUT-related functionality in NGINX. Hence, disabling this edge, e.g., by redirecting it to an aborting instruction, would disable support for the PUT method in NGINX without requiring the identification of all code blocks used in its implementation. *Indirect branches*, such as indirect calls, represent another popular way for activating functionality. Usually, code before the indirect branch points a function pointer to the code implementing the required functionality, which is then later called using that function pointer. For example, to process an image in the IMAGEMAGICK viewer and editor, the appropriate module is invoked through a dispatch table, which contains one function pointer per image type (see Listing 1 in the appendix).

We build on the above observations to accomplish our objective of disabling unwanted *features* ($\mathcal{F}$) in binaries. The approach entails identifying the first control-flow edge, which we term *feature-specific edge* (*FS-edge*), that regulates an undesirable feature and blocking it. Analogous to previous studies [4, 15], we concentrate on deactivating functionality that is not always executed but, instead, requires specific *inputs* for activation. To provide clarity regarding our scope, we consider any data that can be utilized by the program as inputs. Particularly, we focus on the following inputs **whose value corresponds to distinct features**: command-line options, network-protocol and file-format fields, configuration variables stored in files, shell environment variables, and GUI element clicks. In the remainder of this paper, inputs that lead to the activation of an unwanted feature $\mathcal{F}$ are denoted as $I_\mathcal{F}$, whereas other inputs are referred to as $\neg I_\mathcal{F}$.

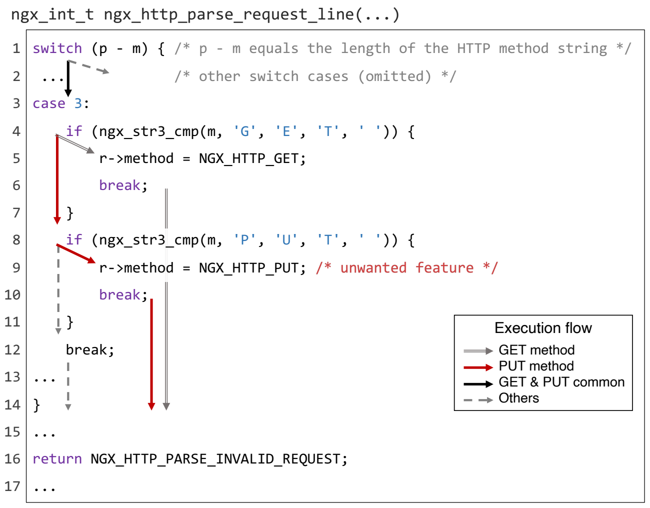

```
ngx_int_t ngx_http_parse_request_line(...)
 1 switch (p - m) { /* p - m equals the length of the HTTP method string */
 2  ...                /* other switch cases (omitted) */
 3 case 3:
 4     if (ngx_str3_cmp(m, 'G', 'E', 'T', ' ')) {
 5         r->method = NGX_HTTP_GET;
 6         break;
 7     }
 8     if (ngx_str3_cmp(m, 'P', 'U', 'T', ' ')) {
 9         r->method = NGX_HTTP_PUT; /* unwanted feature */
10         break;
11     }
12     break;
13 ...
14 }
15 ...
16 return NGX_HTTP_PARSE_INVALID_REQUEST;
17 ...
```

Figure 3: Code snippet from the HTTP method parser of NGINX v1.3.9 for checking the method of a request. The edge $8 \rightarrow 9$ **controls the activation of the PUT-method functionality, which in this instance, is an unwanted feature.**

3.2 F-DETECTOR: Disabling Unwanted Features

To detect the *FS-edge* corresponding to an unwanted feature, we develop F-DETECTOR. F-DETECTOR introduces a set of heuristics to identify the *FS-edge* that can disable $\mathcal{F}$ in binary programs. The heuristics operate on both statically and dynamically collected data, such as the program's control-flow graph (CFG) and execution traces. Figure 4 highlights its design.

Preparing Test Cases We collect execution traces using test cases from two groups: $I_\mathcal{F}$ and $\neg I_\mathcal{F}$. By analyzing the differences between the two groups, we can greatly reduce the search space for the *FS-edge*, as it is bound to be an edge that behaved differently based on the test case group. However, as prior works have cautioned us, randomly selecting test cases can lead to problems. Through experimentation, we found that the edge search space tends to become smaller, when inputs in $I_\mathcal{F}$ and $\neg I_\mathcal{F}$ are similar.

We incorporate the above finding in F-DETECTOR by introducing a set of guidelines for selecting $\neg I_\mathcal{F}$ test cases based on $I_\mathcal{F}$, through *minimal mutation* $\mathcal{M}()$. It involves making small, directed changes to key parts of the input. For instance, assuming $I_\mathcal{F}$ includes the HTTP request `<PUT /test.html HTTP/1.1>` to activate the unwanted PUT method from Fig. 3, our minimal-mutation strategy dictates that we should only replace the `PUT` field with other valid options to generate $\neg I_\mathcal{F}$, such as `<GET /test.html HTTP/1.1>` and `<POST /test.html HTTP/1.1>`. We have also developed similar guides for applying this strategy on the other types of inputs that F-DETECTOR handles, summarized in Table 2. Moreover, we use this process to produce multiple different sets of $\neg I_\mathcal{F}$, allowing us to apply the detection algorithm multiple times. Different $\neg I_\mathcal{F}$ can produce divergent execution traces, which help us avoid debloating errors.

Detecting *FS-edges* The detection process is applied on each pair of $I_\mathcal{F}$ (one) - $\neg I_\mathcal{F}$ (multiple) traces and includes the following steps:

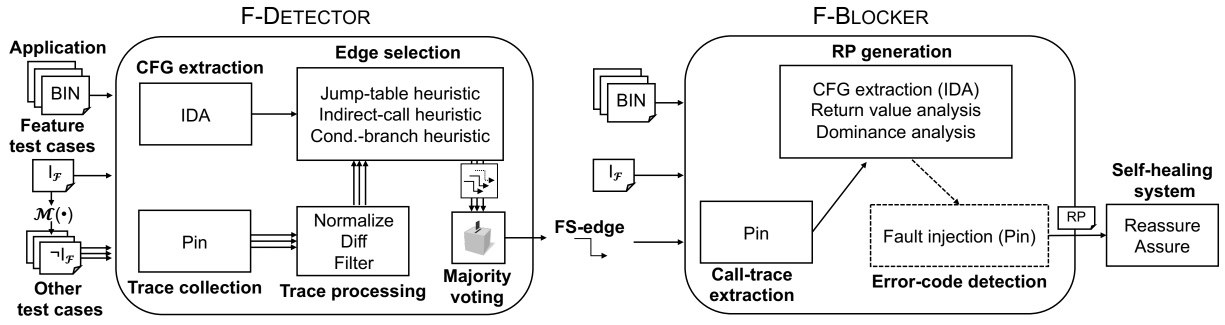


Figure 4: Approach overview. Given a set of test cases, including both $I_\mathcal{F}$ and $\neg I_\mathcal{F}$ inputs, F-DETECTOR attempts to detect the control-flow edge (*FS-edge*) responsible for activating a targeted unwanted feature $\mathcal{F}$. F-BLOCKER generates a rescue points (RP) for an *FS-edge*, which can be used by a software self-healing system to disable $\mathcal{F}$ without affecting survivability.

❶ keep edges taken with all test cases in $I_\mathcal{F}$ but never with $\neg I_\mathcal{F}$, to focus on $\mathcal{F}$ without avoid affecting other features;
❷ eliminate edges from utility functions like `strcmp` in `libc` that support a wide-range of features;
❸ discard edges that do not uniquely control the execution of their destination, e.g., conditional branches (`cbr`) leading to code that is also reachable through other paths. Similar heuristics apply for other types of edges (§4) to avoid affecting other features. The remaining edges are considered *FS-edge* candidates;
❹ select the earliest candidate *FS-edge* in the trace to block all feature-related code. If the edge is part of a chain of `cbr`-based candidates, pick the deepest one instead to accommodate complex condition checks, where the shallower checks control feature groups and the deeper ones control individual features.

How does our algorithm work with the example in Fig. 3? If the PUT method is unwanted, $I_\mathcal{F}$ will include a PUT-method request and we can use a GET-method request as $\neg I_\mathcal{F}$. The execution traces collected will include the following conditional branches:

$$\begin{array}{ll} \text{PUT:} & 1 \longrightarrow 3,4 \longrightarrow 8 \longrightarrow 9,10 \longrightarrow 14 \\ \text{GET:} & 1 \longrightarrow 3,4 \longrightarrow 5,6 \longrightarrow 14 \end{array}$$

Our algorithm will exclude $1 \longrightarrow 3,4$, following ❶, and $9,10 \longrightarrow 14$, following ❸. Edge $3,4 \longrightarrow 8$ is *initially* picked because it is the earliest in the trace. Finally, we decide that the *FS-edge* is $8 \longrightarrow 9$, as it is chained after $3,4 \longrightarrow 8$. Blocking this edge off can disable the PUT method without hurting any other.

The algorithm is robust and will still work if a different method like POST is used in $\neg I_\mathcal{F}$. In that case, the `switch` would jump into another location, not shown in the figure. $1 \longrightarrow 3,4$ would be initially picked due to ❷-❹. Finally, ❹ would pick edge $8 \longrightarrow 9$ because it is the last edge in a chain of valid conditional branches.

Majority Voting on *FS-edge* F-DETECTOR incorporates another mechanism, *majority voting*, to mitigate potential errors in *FS-edge* detection caused by noise in the execution traces. For example, if $\neg I_\mathcal{F}$ is significantly different from the corresponding $I_\mathcal{F}$ it was derived from. By producing multiple *FS-edges*, using different $\neg I_\mathcal{F}$ sets, we can pick the most frequently detected *FS-edge*. F-DETECTOR can also refuse to emit an *FS-edge*, if multiple candidates are found, to avoid the over- and under-debloating issues described in §2. However, the *FS-edge* candidates could still be used to guide analysts and help them discover the correct *FS-edge* manually.

3.3 F-BLOCKER: Exploring Survivability

Given an *FS-edge*, we can disable $\mathcal{F}$ by overwriting the *FS-edge*'s destination with a single-byte instruction, like `int3`, when traversing the edge. This approach is more robust than overwriting large sections of code but may still result in data loss. *How can we provide continuity of service when the disabled functionality is triggered?* To address this, we leverage techniques from software self-healing [36]. Specifically, we introduce *rescue points* (RP), functions that return an error code handled by the application. When an unwanted feature activates, we raise a virtual error, restore execution state to an appropriate RP, and return a valid error code, which will be handled gracefully. F-BLOCKER uses analyses to find a suitable RP. This can then be used with existing self-healing systems like ASSURE [36] and REASSURE [26].

In the example of Fig. 3, the unwanted feature (PUT method) is contained in `ngx_http_parse_request_line()`, which we can use as a RP. Upon entry to the RP, a checkpoint of the process or system state is created. Traversing the *FS-edge* ($8 \longrightarrow 9$) will trigger a fault, causing a rollback to the checkpoint state. Finally, the RP will return with the valid error code `NGX_HTTP_PARSE_-INVALID_REQUEST` to its caller so that NGINX can keep operating. If the *FS-edge* is not hit, the checkpoint state is released upon return.

4 F-DETECTOR

Figure 4 shows an overview of F-DETECTOR. To disable a given feature $\mathcal{F}$, it requires three inputs: application binaries, a set of test cases $I_\mathcal{F}$ that activate $\mathcal{F}$, and multiple sets of $\neg I_\mathcal{F}$ produced by minimally altering $I_\mathcal{F}$, that do not activate $\mathcal{F}$. F-DETECTOR then uses these inputs to determine multiple *FS-edge* candidates, one for each $\neg I_\mathcal{F}$. It then uses *majority voting* among the candidates to select a single *FS-edge*. The rest of this section describes each component of F-DETECTOR in detail.

4.1 Minimal Mutation of Feature Inputs $I_\mathcal{F}$

F-DETECTOR needs both inputs that trigger an unwanted feature ($I_\mathcal{F}$) and inputs that do not ($\neg I_\mathcal{F}$) for tracing. Users can prepare $I_\mathcal{F}$ by selecting a small set of random test cases that activate the unwanted feature. To prepare $\neg I_\mathcal{F}$, a practical approach is to *minimally mutate* $I_\mathcal{F}$ to prevent $\mathcal{F}$ from activating. Users need to provide only a few mutations of $I_\mathcal{F}$ (as few as three mutations based on experiments

Table 2: Minimal-mutation guidelines summary for generating $\neg I_{\mathcal{F}}$ based on $I_{\mathcal{F}}$ and input type. The user will typically pick one type of mutation depending on the type of the feature.

$\mathcal{F}$ Activation Method	Example Test Case ($I_{\mathcal{F}}$)	Guideline	Example Results ($\neg I_{\mathcal{F}}$)
Command-line option	`zip f.zip file -T -TT=val`	Replace option	`-TT` →{`-UN`, `-bs`, `-Z`, …}
Protocol field	`PUT /test.html HTTP/1.1`	Replace keyword	`PUT` →{`GET`, `POST`, …}
File format	`display im.gif`	Convert file	`im.gif` →{`im.jpg`, `im.png`, …}
Configuration variable	`perl_startup = do '/etc/ex.pl'`	Remove option	
Environment variable	`env x='() { :;};'`	Change assignment	`'() { :;};'` →{`' '`, `1`, …}
GUI actions	Click on action A_f under menu M_j	Replace with action under M_j	A_f →{ A_i, for $i \neq f$ }

in §6). For example, to disable the HTTP PUT method in Nginx (Fig. 3), users can create a single PUT request using a utility like `curl`. They can then replace the method passed to the utility with other valid methods to generate $I_{\mathcal{F}}$ as follows:

```
curl -X PUT    http://localhost/file
         ↓
curl -X POST   http://localhost/file
curl -X MOVE   http://localhost/file
curl -X DELETE http://localhost/file
```

We established guidelines on how popular input types handled by F-detector can be minimally mutated by analyzing how various applications handle them. These guidelines are summarized in Table 2 and described in detail below:

Command-line options In command-line programs, options are frequently employed to activate specific functionalities. Usually, a *parser* processes them and updates the program's state, e.g., by setting a variable. We can generate at least two test cases in $I_{\mathcal{F}}$ by using both short and long versions of an option (e.g., `-R` and `--recursive` in `chown`), when available. We minimally mutate them by replacing them with other similar options without modifying anything else, allowing for easy generation of multiple sets of $\neg I_{\mathcal{F}}$.

Protocol fields Many server features are activated based on the protocol field values in received requests (e.g., the PUT method in HTTP). We minimally mutate the single-input $I_{\mathcal{F}}$ by replacing the protocol field with other valid values to generate multiple $\neg I_{\mathcal{F}}$. We avoid modifications to the common parts of the request, unless they are mandated by the new field value.

File formats Applications handling various file types activate functionality based on file format or specific fields within it. For example, image viewer applications support multiple image file types and each of them could be considered as a separate feature. In $I_{\mathcal{F}}$, we can include one or more files of the unwanted format, and minimally mutate them by converting them to other formats. Depending on the format and conversion capabilities, we can generate multiple $\neg I_{\mathcal{F}}$ test cases for each $I_{\mathcal{F}}$.

Configuration variables Variables in configuration files can also control the use of a feature. For instance, the variable `perl_at_start` enables the Perl interpreter in the Exim mailer and runs the script assigned to the variable. We minimally mutate $I_{\mathcal{F}}$ to (multiple) $\neg I_{\mathcal{F}}$ by removing the variable, or replacing the value assigned to it with other valid options.

Environment variables Can be treated similarly to the above.

GUI actions In GUI applications, user actions like keystrokes or mouse clicks trigger features. Usually, through a a callback from the graphical framework in-use into application code implementing the requested functionality. $I_{\mathcal{F}}$ should include inputs corresponding to various activation methods, like clicking on a menu item and using its shortcut. We minimally mutate menu-item clicks by clicking on a different item under the same menu and use a different keyboard shortcut for shortcuts. We can easily generate different sets of $\neg I_{\mathcal{F}}$ for most GUI applications as they usually include numerous actions.

4.2 Execution-Trace Collection and Processing

F-detector captures execution traces of the application, including the addresses of executed basic blocks (BBLs), which are sequences of instructions ending in a control-transfer instruction. By comparing $I_{\mathcal{F}}$ and $\neg I_{\mathcal{F}}$ traces, we aim to identify a small set of control-flow edges, including the *FS-edge*, that satisfy two conditions: they are present in every $I_{\mathcal{F}}$ trace and never in any $\neg I_{\mathcal{F}}$ trace.

Trace Normalization Traces may have multiple sub-traces, one for each thread of execution, identified by the thread ID *tid*. The traces are normalized by recording the unique control-flow transitions of each sub-trace as a source–destination pair of BBLs (*src*-*dst*). The position (*pos*) and the number of appearances (*num*) of each BBL in the sub-trace are also included. The normalized traces are merged into a single trace with tuples in the form of $(src, dst, tid, pos, num)$, representing unique edges across threads. *tid*, *pos*, and *num* correspond to those of the thread sub-trace in which they first appeared. We refer to the normalized traces as *profiles* in the rest of this section.

Profile Diffing To obtain a set of control-flow edges that includes the desired *FS-edge*, we compare the collected profiles. We start by generating set C as the intersection of all (src, dst) pairs in $I_{\mathcal{F}}$ profiles, which represents the CFG edges consistently taken in all executions where the feature is activated. We then generate set E as the union of all edges in $\neg I_{\mathcal{F}}$ profiles. Subtracting E from C gives us a set of edges that do not appear in $\neg I_{\mathcal{F}}$ profiles.

Utility-Function Filtering Applications often rely on utility functions, such as string-comparison functions, from libraries such as `libc`. These functions are typically not directly related to any $\mathcal{F}$, so the *FS-edge* is unlikely to be located within them. Therefore, we filter out ranges that correspond to utility libraries, including user-configured libraries, to exclude such edges. We also eliminate built-in utility functions, which are called frequently even for basic workloads, by discarding edges that occur multiple times. For instance, in the ImageMagick application, `CopyMagickString()` is called multiple times and contains a loop that causes internal edges to appear multiple times, leading to their exclusion.

4.3 FS-Edge Detection

We use the heuristics identified in the previous stage to detect the *FS-edge* based on how programs usually activate functionality. This overcomes the limitation of working with traces obtained with few inputs, which may contain edges associated with other functionality besides $\mathcal{F}$. We group consecutively executed edges into packs, where each pack contains edges where the destination BBL of the first edge is the source BBL of the second. We search for the earliest pack containing an *FS-edge* based on our heuristics, going over packs in order from earlier to later executed edges. This method exploits the fact that *FS-edge* usually appear early in the diffed executions with $I_{\mathcal{F}}$ and $\neg I_{\mathcal{F}}$.

Detection Heuristics *FS-edges* correspond to conditional program control flows, where $\mathcal{F}$-related code is executed conditionally to input. C and C++ programs use three common mechanisms to implement such logic:

- `if-then-else` statements: in binary code, they are implemented by a conditional branch (`cbr`) instruction, like `je` in x86 binaries.
- `switch` statements: they can be implemented either as a sequence of `cbr` or using an indirect jump (`ijmp`) instruction (e.g., `jmp rax` in x86) using pointers from a compiler-generated jump table, containing one entry per switch case.
- Function pointers: they are implemented as indirect calls (`icall`) or jumps (`ijmp`), often using a developer-provided function table.

We have developed three heuristics based on the above constructs, which we apply on each pack of edges. If no *FS-edge* is detected, we proceed to the next pack, and if none are left we emit no *FS-edge*. Our heuristics are the following:

(i) *Jump-table heuristic*: when an `ijmp` corresponds to a `switch` jump table, we treat it as a `cbr`. This targets applications using a switch statement to conditionally activate $\mathcal{F}$ via a jump table. Our example in Fig.3 contains one such `switch` (line 1) to check different method-string lengths. We use the IDA Pro disassembler[11] to analyze the code and detect jump tables and their corresponding `ijmp`.

(ii) *Indirect-call heuristic*: if the pack starts with an `icall` or a non-switch `ijmp` edge, we select that as *FS-edge*. The heuristic captures applications using a dispatch table containing pointers to functions associated with different features. Listing 1 shows one such use in the ImageMagick application for loading different image-format processing modules.

(iii) *Conditional-branch heuristic*: if the pack starts with a `cbr`, we consider the application may be using an `if-then-else` or `switch` to activate $\mathcal{F}$. Applications often link multiple BBLs, testing for increasingly specialized conditions. An example of such a pattern exists in our Nginx example (Fig.3), where a `switch` is used to first test for method-string length (1⟶3), followed by `if-then` statements testing for specific method names (4⟶8⟶ 9). To handle such cases, we recursively process all edges in the pack until a stop condition is reached. If at least one `cbr` was found before then, it is returned as *FS-edge*. The stop conditions are:
   - the end of the edge pack is reached;
   - a function-call or return edge is encountered, terminating the chain of `cbr`;
   - an edge to a BBL with multiple incoming edges occurs. This rule aims to exclude branches that lead to code which could also be executed through other paths. Such paths may exist, even if they have not been observed during tracing. For example, in Fig. 3 the `break` on line 10 leads to a jump to the end of the `switch` statement, which is also accessible by other cases. To identify such BBLs, we utilize IDA to statically obtain the partial CFG of the application and determine if there are multiple incoming edges. The goal of this rule is to select a `cbr`-based *FS-edge* that uniquely controls the execution of its destination and avoid under- and over-debloating.

FS-Edge Detection in the Presence of Threads To handle the challenge of absolute ordering in multi-threaded applications, our algorithm requires the edges in each profile to be ordered. However, achieving absolute ordering on multi-core architectures where threads run in parallel is difficult. To overcome this issue, we apply our approach to each thread's profile individually, which may result in identifying one *FS-edge* per thread. We assume that the *FS-edge* executed first caused the subsequent ones. To select it, we re-run the application with one of the inputs in $I_{\mathcal{F}}$ while instrumenting it to record the traversal order of *FS-edges*.

Majority Voting F-detector applies the *FS-edge* detection algorithm multiple times with both the $I_{\mathcal{F}}$ set and each of the $\neg I_{\mathcal{F}}$ sets, resulting in several *FS-edge* candidates. To select *one FS-edge*, we have two options: (i) be strict and only consider the *FS-edge* if all candidates agree (unanimous decision), or (ii) use majority voting and select the *FS-edge* (if any) that the majority of runs produced. In our evaluation, we use option (ii) as it produces better results.

5 F-blocker

F-blocker automatically defines a rescue point to recover from unexpected errors when attempting to execute $\mathcal{F}$. This point can be used with a software self-healing system like ASSURE [36] or REASSURE [26]. The RP function must satisfy the following criteria: **P1**: all possible paths leading to the *FS-edge* include the function, so we can always "rescue" the application; **P2**: the function returns an error code handled by its callers; **P3**: it is near the *FS-edge* to reduce overhead. Fig. 4 provides an overview of F-blocker's components, which are discussed in detail in the following sections.

Call-Trace Extraction We run the application using the $I_{\mathcal{F}}$ inputs used by F-detector, recording function caller-callee pairs, function returns and potential return values, active memory mappings upon return, system calls and their return values. Upon hitting the *FS-edge*, we also record the call stack at that point and terminate the run.

5.1 Rescue-Point Generation

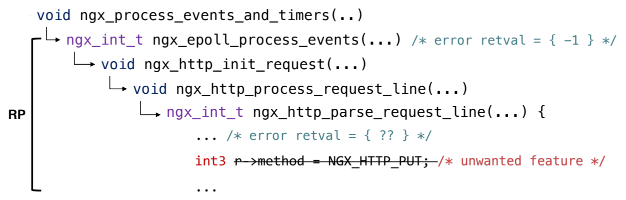
Dominance Analysis To satisfy **P1**, the RP must be one of the functions in the call stack obtained in the first step. To identify eligible functions, we extract the application's CFG to examine their relations. Functions that dominate the one containing the *FS-edge* (meaning all execution paths go through them) are RP candidates. If a function in the call stack is address taken (AT), meaning the program has a reference to it, we instead rely on call-trace data to assign callers (usually one) based on run-time observations. Although this may underestimate callers, if one of the callers of an AT function is eventually chosen as an RP, manual analysis can be used to verify that **P1** is satisfied before deployment.

Return-Value Analysis To satisfy **P2**, we need to automatically determine whether the eligible functions (i) return a value (i.e., not `void`), (ii) which values are associated with errors, and (iii) whether these errors are handled by the application. Binary applications use calling conventions, such as using the `EAX`/`RAX` registers in x86 architectures to return values. We conduct a static analysis of the application to determine (i) and (iii) as follows:

- For functions that are not address-taken, we analyze all their callers to determine whether `EAX`/`RAX` is used before being set, right after a call returns, indicating the value is not ignored.
- For AT functions, since we cannot discover all callers, we instead use the callee's code. Specifically, we examine if every execution path within the function sets `EAX`/`RAX` without using its value before returning, which indicates that the function always returns a value.

To establish return values indicating errors, previous research relied on static analysis [12, 42]. However, these approaches can undermine the safety of our system, so instead we opt for a *dynamic* approach, which is inspired by two observations. Functions that return pointers often return `NULL` to indicate an error, and functions that issue system calls often test and return an error code to their callers. Based on these observations and the values we observe during tracing, we treat a return value as a pointer and `NULL` a valid error code, if it lies in the address range of mapped memory. Otherwise, we assume that the function returns integers and use the method below to determine error codes.

Error-Code Detection To detect error codes for functions that return integers and make system calls, we use fault injection at the system call level. Specifically, we re-run the application with $I_{\mathcal{F}}$ and deliberately fail all system calls in the target function. If the return value changes compared to the previous run, we consider this new value as an error code.



```
void ngx_process_events_and_timers(..)
└→ ngx_int_t ngx_epoll_process_events(...) /* error retval = { -1 } */
    └→ void ngx_http_init_request(...)
        └→ void ngx_http_process_request_line(...)
            └→ ngx_int_t ngx_http_parse_request_line(...) {
                   ... /* error retval = { ?? } */
                   int3 r->method = NGX_HTTP_PUT; /* unwanted feature */
                   ...
```

Figure 5: Rescue Point for Disabling Nginx's PUT method.

RP Selection After collecting all functions that satisfy **P1** and **P2**, we choose the one closest to the *FS-edge* and its associated error code, which becomes the rescue point (RP). For instance, to disable Nginx's PUT method, F-blocker selects function `ngx_epoll_process_events` and return value `-1` as the rescue point, as shown in Fig. 5. This function satisfies all three properties (**P1-P3**). Although `ngx_http_parse_request_line` would be the ideal RP, it does not make any system calls, which prevents us from inferring the return value used to indicate an error to its callers.

6 Implementation and Evaluation

Our system is implemented* on top of Intel's Pin [20] and IDA Pro [11] using ≈4K lines of C++ code and 700 lines of Python code. Currently, F-detector supports Linux platforms, but its tools can be extended for Windows platforms. We evaluate F-detector to answer the following questions: (i) *How does it compare with Razor in terms of under- and over-debloating?* (ii) *Can it disable features in larger and more complex applications?* (iii) *Can it reduce the attack surface of software and neutralize vulnerabilities?* (iv) *Can bad mutations affect its results?* and (v) *Can F-blocker provide server continuity when unwanted features are triggered?*

*Publicly available on https://github.com/MohamadMansouri/feature-disable.

Table 3: Successfully disabled features in Razor's benchmark suite (coreutils) without over or under-debloating.

Application	Wanted Features	Unwanted Features
bzip2-1.0.5	-compress	-decompress, -test
chown-8.2	-recursive, -no-dereference	-changes, -no-preserve-root, -verbose, -from, -reference
date-8.21	+%c, +%d, +%D, +%F, +9 more	+%A, +%a, +%b, +%B, +6 more
grep-2.19	-regexp, -extended-regexp	-basic-regexp, -perl-regexp, -word-regexp, -line-regexp, +4 more
gzip-1.2.4	-compress	-decompress, -test
mkdir-5.2.1	-mode, -parents	-verbose
rm-8.4	-recursive, -force, -interactive	-one-file-system, -no-preserve-root, -verbose
sort-8.16	-reverse, -unique, -stable, -zero-terminated	-ignore-case, -month-sort, -numeric-sort, -random-source, +6 more
tar-1.14	-create	-list, -extract, -compare, -append, +3 more
uniq-8.16	-count, -repeated, -skip-fields, +4 more	-zero-terminated

6.1 Comparison with Razor

We test F-detector on the same benchmark programs as Razor to determine, if we can remove features without similar issues. We consider the features used to train Razor as wanted features and, based on them, select some of the excluded ones as unwanted features to disable (listed in Table 3). We use two mutations for each unwanted feature and manually verify that the *FS-edge* detected fully disables it without affecting any of the wanted features. We find that F-detector managed to disable all unwanted features without any over- or under-debloating problems.

6.2 Disabling Features in Larger Applications

We evaluate F-detector's ability to disable features in larger software using 8 popular, real-world Linux applications that span a wide spectrum of families: servers (Nginx v1.3.9, ProFTPD v1.3.5e, and Exim v4.86), utilities (Zip v3.0 and Exiv2 v0.27.1.19), GUI applications (ImageMagick v7.0.9 and Evince v3.22.1), and a shell (Bash v.4.3). We target 40 prominent features of different types that (i) correspond to various functionalities and services; (ii) are activated by different types of inputs (command-line options, files, network data, environment variables, configurations, and UI clicks); and (iii) are associated with various types of vulnerabilities.

For each feature, we run two set of experiments using two and three mutations to establish their effect. In each experiment, we

Table 4: Evaluation of F-DETECTOR with larger applications. All features can be disabled by an *FS-edge*. TP is the number of tests where the detected *FS-edge* fully disables the unwanted feature without hurting other features; FN is the number of tests where majority cannot be reached and no *FS-edge* is emitted; FP is the number of tests where the detected *FS-edge* cannot disable the unwanted feature or hurts other features. Features with identical results are collapsed into the same row/label.

Application	$\mathcal{F}$ Type	$\mathcal{F}$ (Feature)	$I_{\mathcal{F}}$ Size	$M = 2$	$M = 3$	CVEs
				TP / FN / FP	TP / FN / FP	
ImageMagick (7.0.9)	GUI button	Crop, Chop, Flop, Flip, Rotate, Shear	1	10 / 0 / 0	10 / 0 / 0	
Evince (3.22.1)	GUI button	Print, Open, Save, Copy, Properties	2	6 / 0 / 0	4 / 0 / 0	
ImageMagick (7.0.9)	Formatted input – file (image file support)	TIFF	2	1,584 / 0 / 0	528 / 0 / 0	2019-13136, 2019-15141
		PNG		16,068 / 0 / 0	5,356 / 0 / 0	
		JPEG		9 / 0 / 0	3 / 0 / 0	
		GIF		55 / 29 / 0	21 / 7 / 0	
Nginx (1.3.9)	Formatted input – network (HTTP methods and options)	Chunked Transfer Encoding	1	1 / 0 / 0	—	2013-2028
		GET, MOVE, POST, PUT		3 / 0 / 0	1 / 0 / 0	
ProFTPD (1.3.5e)	Formatted input – network (FTP commands)	CHGRP, CHMOD	1	1 / 2 / 0	1 / 0 / 0	
		CPTO		1 / 2 / 0	1 / 0 / 0	2015-3306
		CPFR		3 / 0 / 0	1 / 0 / 0	2015-3306
Exiv2 (0.27.1.19)	Cmd. line option	Insert, Remove, Print, Extract, Rename	1	12 / 6 / 0	8 / 4 / 0	
Zip (3.0)	Cmd. line option	`-ds`, `-UN`, `-b`, +5 more	2	28 / 0 / 0	56 / 0 / 0	
		`-TT` (unzip command)		28 / 0 / 0	56 / 0 / 0	priviledge escalation [3]
Exim (4.86)	Config file & cmd. line option	`perl_at_start`, `-ps`	2	15 / 0 / 0	20 / 0 / 0	2016-1531
Bash (4.3)	Environment variable	Function definition (e.g., `x="() { :; }"`)	1	1 / 0 / 0	—	2014-6271

target one feature for removal, mutating it to the other features listed in Table 4. For instance, for ImageMagick as a command-line utility, a TIFF image is transformed to the other image types: `tiff` →{`jpg`, `png`, `gif`}. We conduct an experiment *for each* combination of mutations, for example, in this case where there are three possible mutations and $M = 2$, there are $C(3, 2) = 3$ combinations for each $I_{\mathcal{F}}$. As inputs, we use images from the test suites of libraries, specifically, we use 33 `libTIFF`, 104 `libpng`, 3 `libjpeg`, and 3 `giflib` images. Finally, we use two images in $I_{\mathcal{F}}$ and also try out all possible combinations images, so for TIFF images we get $C(33, 2) = 528$ different $I_{\mathcal{F}}$. The approach is similar for the rest of the applications, but for inputs we manually click on buttons activating the targeted feature in GUI applications, we use the same file while mutating command-line options in Zip and Exiv2, we use the `curl` and `ftp` utilities to interact with Nginx and ProFTPD, and we simply execute Exim and Bash.

Table 4 lists the applications and features, and summarizes the results of our experiments. We classify each experiment as a True Positive (TP), False Positive (FP), True Negative (TN), or False Negative (FN), with a positive corresponding to F-detector reaching consensus on an *FS-edge*. To determine if it is a *True* or *False FS-edge*, we manually inspect application code to verify that the *FS-edge* detected stops the execution of the feature and it does not affect others. A FN corresponds to an experiment where F-detector is inconclusive (voting did not agree on an *FS-edge*), while such an edge exists. A TN corresponds to a feature that cannot be disabled by an *FS-edge* and F-blocker correctly does not emit an edge either, which we did not encounter.

Inconclusive Cases (FN) F-detector was not able to find an *FS-edge* with $M = 2$ for three applications. We discuss the reasons below:

ImageMagick Two of the GIF images in the test suite included animations, which when converted to PNG and TIFF retained GIF-formatted data. Consequently, $\neg I_{\mathcal{F}}$ included inputs that still activated GIF-related functionality, so $\neg I_{\mathcal{F}}$ still used the GIF feature.

ProFTPD The selected features belonged to two command categories [40]: *Direct File Duplication* (`CPFR`/`CPTO`) and *Owner or Group Change* (`CHGRP`/`CHMOD`). The first group are sub-commands of the `SITE` command and ProFTPD also parses them as a subgroup of commands. By using a $\neg I_{\mathcal{F}}$ outside the `SITE`-group of features to disable a feature within the group, essentially, we are causing F-detector to target all `SITE` commands in one of the mutations.

Exiv2 Exiv2 provides three different ways to activate options. For example, to insert metadata one can use option `insert`, `in`, or `-i`. Exiv2 uses a separate parser for dashed options, leading to a similar grouping problem as ProFTPD. Using all aliases of an option in $I_{\mathcal{F}}$ would eliminate this issue (more in §7).

Overall, our experiments indicate that F-detector produces correct results for a variety of applications and features. Increasing the number of mutations M also helps resolve all FNs, even in the cases where inputs are not selected carefully. As an alternative, we can resort to manual analysis of the candidate *FS-edges* produced by F-detector to determine of one of them is correct.

6.3 Security Benefits of Feature Removal

We evaluate F-detector's vulnerability-mitigation capabilities by examining known vulnerabilities present in our application set.

CoreUtils For CoreUtils, we use the same vulnerabilities as Razor (only the ones that affect the utilities) with F-detector/F-blocker and compare the two. To determine if F-blocker actually mitigates a vulnerability, we manually analyze applications to validate the reachability of vulnerable code or functionality after blocking the *FS-edge*. Table 5 lists the vulnerable feature for each utility and the results of our evaluation.

Based on the results reported by its authors [27], Razor fails to mitigate 6 of the 10 vulnerabilities. Note that for Razor a vulnerability is considered mitigated, only if the function containing it is erased from the binary, even if all the paths leading to it have been debloated, so it is possible that they under-approximate the vulnerabilities mitigated. F-blocker is able to mitigate all vulnerabilities except CVE-2014-9471, which resides in the core functionality of `date` and is associated with multiple features. It corresponds to a denial-of-service vulnerability, where an attacker can provide a malicious crafted date to crash the program. The vulnerability lies in function `parse_datetime`, which is used by many features, like `-date`, `-file`, `-TZ`, etc., and Razor was also unable to disable it. We further discuss the mitigated vulnerabilities, organized by type, below:

Race condition CVE-2017-18018, CVE-2005-1039, and CVE-2015-1865 are time-of-check, time-of-use (TOCTOU) vulnerabilities that affect `chown`, `mkdir`, and `rm`, respectively. They are all associated with particular functionality of these utilities and can be mitigated by F-blocker by disabling it. Specifically, disabling recursive directory traversal in `chmod` and `rm`, and creating a directory with specific permissions with `mkdir`.

Memory corruption CVE-2010-0405, CVE-2015-1345, CVE-2010-0001, and CVE-2009-2624 are memory-corruption vulnerabilities, with the first and third introduced by an integer overflow. The vulnerabilities are contained within specific functions of the utilities that are only reachable when certain features are activated, like decompression for `gzip` and `bzip2`, and the `-fixed-strings` option in `grep`. F-blocker is able to mitigate them by disabling these options. For interested readers, the corresponding functions made unreachable are: `bzip2`→`BZ2_decompress()`, `grep`→`bmexec_trans()`, `gzip`→`huft_build()` and `unlzw()`.

Directory traversal CVE-2005-1228 is a directory traversal vulnerability in `gzip` that allows an attacker to write to arbitrary directories by using '`..`' in the filename passed to the `-name` option. F-detector disables the `-name` option in `gzip`'s parser, mitigating this vulnerability.

Overall, F-detector and F-blocker can target and disable specific features by design, unlike Razor which despite erasing more code, allows vulnerabilities to persist due to the heuristics working to avoid over-debloating.

Larger Applications We conduct the same experiment for the CVEs and applications listed in Table 4. We chose these CVEs because they reside in a specific feature of the applications that we previously selected. To determine if F-blocker mitigates a vulnerability, we performed the same kind of manual analysis we did for the CoreUtils vulnerabilities. Moreover, we used the proof-of-concept inputs accompanying the CVEs, which trigger the vulnerable code, to ensure that we did not falter during the manual analysis, as these applications are significantly larger. In all cases, F-blocker caused the application exit, before it reached vulnerable code, mitigating all vulnerabilities. Below, we further discuss each one:

CVE-2013-2028 The vulnerability is a stack buffer overflow in Nginx's `ngx_http_parse_chunked()` [32]. This function is called by the four functions (marked by ①-④ in Listing 5), *only* when `r->headers_in.chunked` is *true*. The only location setting that field to true is the destination basic block of the *FS-edge* detected by F-detector (shown in Listing 4).

CVE-2019-13136 and CVE-2019-15141 Both these vulnerability lie in the `libTiff` module of ImageMagick. F-detector detects the *FS-edge* as an indirect call to `RegisterTIFFImage` (see Listing 1), which loads `libTiff` at run time, therefore neutralizing the vulnerability.

CVE-2015-3306 This ProFTPD vulnerability allows unauthenticated users to copy files around on the server using commands `CPFR` and `CPTO`. Since F-blocker can disable both of them, using the *FS-edge* in Listing 9, the vulnerability is mitigated.

CVE-2016-1531 This vulnerability allows users to run a Perl script with elevated privileges, when Exim is installed with setuid on, by using the `perl_startup` option. The *FS-edge* detected blocks access to the only call site of `init_perl`, hence, mitigating the issue (see Listing 7).

CVE-2014-6271 This Bash vulnerability allows arbitrary code execution through defining a function in an environment variable. This particular functionality is implemented in `initialize_shell_variables()`, which contains the *FS-edge* detected by F-detector (see Listing 8). By blocking the environment-variable function parser, F-blocker mitigates the issue.

Zip priviledge escalation Zip allows users to replace the command used to decompress files through the option `-unzip-command/-TT`. Thus, running Zip with elevated privileges allows attackers to run arbitrary commands. The variable `unzip_command` is only assigned during options parsing, which matches the *FS-edge* blocked by F-blocker (see Listing 6).

Table 5: Mitigation of CoreUtils vulnerabilities.

Application	CVE	Vulnerable Feature	Razor	F-detector
`bzip2-1.0.5`	CVE-2010-0405	`-decompress, -test`	✗	✓
`chown-8.2`	CVE-2017-18018	`-recursive/-L`	✓	✓
`date-8.21`	CVE-2014-9471	Multiple	✗	✗
`grep-2.19`	CVE-2015-1345	`-fixed-strings`	✓	✓
`gzip-1.2.4`	CVE-2005-1228	`-name`	✓	✓
	CVE-2009-2624	Type 2 decompression (dynamic huffman code)	✗	✓
	CVE-2010-0001	LZW decompression	✓	✓
`mkdir-5.2.1`	CVE-2005-1039	`-mode`	✗	✓
`rm-8.4`	CVE-2015-1865	`-recursive`	✗	✓
`tar-1.14`	CVE-2016-6321	`-extract`	✗	✓

6.4 Effects of Bad Mutations

To evaluate the effect of mutations, when we have not necessarily adhered to our minimal mutation guidance, we use BusyBox, a tool that combines tiny versions of many common UNIX utilities into a single executable. We evaluate F-detector with each of the 105 features implemented by BusyBox v1.22.0. For each feature, we run an experiment for each possible combination of mutations, so for $M = 3$ we try $C = 182104$ combinations.

For 100/105 features, F-detector reaches a consensus on the correct *FS-edge* in 99.5% (181,104) of the experiments with three mutations. The *FS-edge* is routed on the same instruction because it corresponds to an `icall` to the function implementing the unwanted applet.

However, we encounter significant errors (≈99%) when trying to disable the remaining five features. These correspond to the following applets: `ping`, `traceroute`, `ping6`, `traceroute6`, and `crontab`. Features are grouped into two groups (100|5), based on whether they need to drop SUID privileges. As a result, when trying to disable the five that do not drop privileges, using mutations from the other set (the vast majority of features), F-DETECTOR ends up disabling the keep or drop SUID feature. Increasing the number of mutations will help detect these cases. However, in the case of BusyBox, the vast majority of selected mutations are problametic. Thus, we need to run with more than 100 mutations (this will include at least one of the correct mutation) to detect the error.

Alternatively, a better strategy is to only mutate to similar applets, e.g., file-utilities to file-utilities, network-utilities to network-utilities and so on. Selecting mutations more carefully can significantly reduce errors, if not eradicate them. This experiment shows that consulting manuals and having some familiarity with the software being debloated is important. This is a reasonable expectation for system administrators and similar roles.

6.5 Continuity of Service after Feature Removal

We finally tested whether F-BLOCKER can maintain the continuity of server programs using the generated rescue points with an existing software-self healing system, namely REASSURE [26]. REASSURE, which was made available to us by its authors, implements rescue points over Pin, using log-based checkpoint and rollback. While it is limited in terms of performance and scope of checkpoints, it enabled us to verify the effectiveness of the generated RPs.

We applied F-BLOCKER on the *FS-edge* detected for NGINX's PUT method (other methods can be handled similarly) and for PROFTPD's `CPFT` command. The RP for NGINX is on function `ngx_-epoll_process_events` and returns an error value of `-1`, while for PROFTPD it is on function `copy_cpfr` and it returns a NULL error value. In both cases, when deploying the RPs the fault triggered by the *FS-edges* is correctly virtualized allowing the services to continue processing requests. In particular:

- `ProFTPD` returns error code `500` to the client that issued the invalid command, along with a message that the command cannot be interpreted, and continues accepting new connections.
- `Nginx` stops processing the request and returns to the main serving loop. The user does not receive an error message, but its connection is terminated.

For reference, we list all the RPs generated by F-BLOCKER for all tested applications in Table 6 in the appendix.

7 Limitations and Future Work

Command-line option aliases Command-line utilities usually take various options to customize their operation and access different features. On Linux, options have a short ('-' prefix) and long form ('--' prefix), for instance, `chown -R` is equivalent to `chown --recursive`. F-DETECTOR methods attempt to capture option activation at the parsing staging to disable them before any feature code is run. Other utilities may provide more *aliases* for an option, for example, EXIV2 provides three different ways to activate options: `exiv2 insert`, `exiv2 in`, and `exiv2 -i a` are all equivalent. The unique aspect of EXIV2 is that it uses two separate parser routines to process options. Therefore, if we only used the first two inputs, which use the same parser, the detected *FS-edge* would not disable the third alias of the *insert* option. To avoid such errors, it is important to include all aliases of an option in $I_{\mathcal{F}}$. Interestingly, GUI applications do not suffer from this issue by construction, because the frameworks, they are built upon, are event driven and use indirect calls and dispatch tables.

Problematic Non-Feature Inputs Our experiment with BusyBox revealed that picking $\neg I_{\mathcal{F}}$ indiscriminately can be problematic. By further looking at BusyBox material, we found that system administrators are already aware of the different applet requirements in terms of access rights. BusyBox is a corner case in this respect, as system administrators are aware of which applets require additional permission, but the manual itself does not explicitly differentiate between them. It is possible that system tools may require additional expertise when selecting which mutations to use during trace collection.

Augmenting Traces with Data Flow Information The above limitation could be potentially addressed by also collecting data flow information while collecting traces, for instance, using a data-flow tracking tool, such as LIBDFT [17]. This would help us determine if a branch is using information dependent on mutated inputs. Further research is required to determine how such information can be incorporated into our algorithms.

8 Related Work

Debloating Unreachable Code Various prior works have focused on detecting code that is included in applications, because of the use of shared libraries, packages, etc., but is actually never used and is unreachable during execution. Such code can be eliminated through techniques employing static and dynamic analysis. For example, JRED [14] debloats Java applications and the Java Runtime Environment by performing conservative static analysis on Java bytecode to understand reachability. It relies on rewriting bytecode files to remove unreachable methods and classes. Quach et al. [28] focus on C and C++ applications to eliminate unreachable code from the libraries they use. They analyze applications and libraries to extract their external dependencies and function-call graph (FCG) at compile and link time. The extracted information is utilized at load time by a customized loader to eliminate disconnected and thus unreachable code. Similarly, Nibbler[1] analyzes binary applications to obtain their FCG and library dependencies to produce debloated shared libraries, where unused functions have been erased. BinTrimmer [29] improves the reachability analysis performed by previous binary analysis tools by using a type of value-set analysis (VSA) to refine the extracted control-flow graph (CFG). It offers small improvements over the Angr [35] binary-analysis framework, hence, eliminating more unreachable code.

Debloating unreachable code comes with virtually no overhead; however, it does *not* affect the reachability of known or unknown vulnerabilities in applications, since they are only relevant if located in code used by some part of the application. *Instead*, this work aims to block code paths that are used by applications and can potentially include relevant vulnerabilities. The discussed works can also remove many gadgets that can be used in code-reuse attacks (CRAs), but they are not able to remove all instances of gadgets useful to attackers from all libraries, packages, etc. Consequently, their post-exploitation benefits are application and vulnerability dependent. In this respect, F-DETECTOR is *orthogonal* to gadget reduction approaches.

Dynamic Debloating Works in this line of research dynamically include or exclude library code at run time. BlankIt [25] proposes a debloating technique that selectively activates library functions as they are used by an application. When an API function is invoked, its code, along with all reachable functions, are enabled, while the remaining code is disabled (i.e., debloated). Static analysis of code determines the reachable code for each API function. PacJam [24] is another solution that utilizes static analysis to identify the library dependencies of applications, which it then uninstalls by removing their corresponding Linux packages. It also traces applications using test inputs of to determine the packages used in those scenarios, replacing all other dependencies with mostly-blank, shadow packages. To avoid issues caused by the limited coverage of test cases, PacJam can reinstall a package if its code ends up being used by an application at run time.

Like the studies discussed earlier, BlankIt does not eliminate vulnerabilities from applications, as all reachable code can be loaded on-demand. The same is the case with PacJam when packages are dynamically reinstalled. Not doing so significantly debloats libraries at the package level, but it can lead to application crashes when inputs differ from the test inputs used. As such, it can only be applied when test inputs accurately represent all possible desired inputs. In terms of performance, BlankIt also imposes non-negligible run-time overhead of about 20% on average.

Debloating based on Wanted Features Many past approaches have focused on eliminating unwanted code by disabling all features except a small set of wanted ones. State-of-the-art program reduction tools such as Perses [38] and C-Reduce [30] build upon the concept of delta debugging [21, 41] to minimize the size of a given program while preserving its correctness with respect to a specified property test function. Recently, Chisel [10] has further improved upon this approach by incorporating reinforcement learning. Through trial and error, Chisel builds a model to predict the likelihood of a candidate minimal program passing the property test.

Chisel and our approach share a reliance on input-based specifications. However, Chisel requires comprehensive inputs that activate all desired functionality, which can be difficult to generate. Previous studies have shown that even developer-written tests often achieve low coverage of program functionality [13]. In contrast, our approach requires only a small number of inputs and a set of guided mutations to disable specific functionality. Additionally, Chisel employs statistical methods that may introduce errors. Razor [27], which we extensively discuss in §2, uses heuristics instead of statistical methods to overcome Chisel's limitations, but still results in over- and under-debloating errors. Although our approach also uses heuristics, it focuses on disabling specific application features, requires fewer inputs and avoids the same errors. Finally, it is worth noting that Chisel operates on source code, while Razor and F-DETECTOR operate on binary code, which has its semantic information stripped during compilation.

Disabling Unwanted Features More closely-related to our work are approaches focusing on disabling unwanted features. JCut [15] operates on Java programs and relies on manually-identified *seed methods* that define an unwanted feature. To remove a feature, JCut removes all call sites to the seed methods and all code made redundant by this removal. Similarly, DamGate [4] aims to debloat binaries by associating each function with a feature and blocking features by inserting gates at all possible call sites of the function. A follow-up work [5] employs symbolic execution to improve gate insertion for network-based applications. However, these approaches depend heavily on accurate method or function selection, require extensive developer involvement, and are time-consuming.

Test-based Software Minimization [6, 7] (TBSM) is a technique that removes unwanted functionality based on developer-defined, annotated test cases. TBSM's approach leverages developers' familiarity with tests, which can make defining unwanted functionality more practical than using formal methods or architectural descriptions. TBSM can remove arbitrary functionality from applications, even if it is not directly connected to inputs. However, it does require extensive test cases to be developed, which faces the same challenges we presented earlier.

Landsborough et al. [19] developed several early-stage approaches for removing features. These approaches consider the instruction traces following a specific group of inputs as a feature. To remove unwanted functionality, they collect instruction traces with test inputs for both desired and undesired features. They then rewrite the code that is never reached with *nops*, while also overwriting the code activated by unwanted features but not by desired ones. However, these approaches, as well as TBSM, largely rely on the comprehensiveness of test inputs, which can easily result in accidental removal of functional code.

Neutralizing Known Vulnerabilities TALOS [12] proposes an alternative to feature removal for mitigating vulnerabilities without patching. It operates on source code to create security workarounds for newly disclosed vulnerabilities, by redirecting all execution paths that reach them to builtin error handling code. TALOS was later extended in RVM [42] to also handle binary-only programs. In comparison to these works, our approach is not limited to blocking known vulnerabilities, but can pro-actively disable unnecessary features to protect from undiscovered ones. F-BLOCKER is also more robust at recovering execution, because of its use of state rollback.

9 Conclusion

This paper presents a novel approach to reducing attack surface by disabling unwanted features, without the need for burdensome and extensive user specifications and inputs. Our method involves dynamic tracing and static analysis to identify a single control-flow edge that leads to the unwanted feature and uses error virtualization to safely disable it. This ensures that normal service continues in the target program. We have implemented and evaluated our approach on Linux, successfully disabling 145 features in 9 applications. Our results demonstrate the effectiveness of our method in improving security for a range of binary applications and reveal worthwhile areas for future research.

Acknowledgments

We thank the anonymous reviewers and Juanru Li, our shepherd, for their valuable comments. This work was supported by the Office of Naval Research (ONR) awards N00014-17-1-2788 and N00014-17-1-2787, DARPA award D21AP10116-00, and National Science Foundation (NSF) award CNS-2213727. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, ONR, DARPA, or NSF.

References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (San Juan, Puerto Rico). ACM, USA, 70–83.
[2] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman. 2016. Design and Implementation for Checkpointing of Distributed Resources Using Process-Level Virtualization. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, USA, 402–412.
[3] Raj Chandel. 2019. Linux for Pentester : ZIP Privilege Escalation. https://www.hackingarticles.in/linux-for-pentester-zip-privilege-escalation/.
[4] Y. Chen, T. Lan, and G. Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. ACM, 23–29.
[5] Y. Chen, S. Sun, T. Lan, and G. Venkataramani. 2018. TOSS: Tailoring Online Server Systems Through Binary Feature Customization. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. ACM, 1–7.
[6] A. Christi, A. Groce, and R. Gopinath. 2017. Resource Adaptation via Test-Based Software Minimization. In *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 61–70.
[7] A. Christi, A. Groce, and A. Wellman. 2019. Building Resource Adaptations via Test-Based Software Minimization: Application, Challenges, and Opportunities. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 73–78.
[8] Will Glozer. 2019. WRK: Modern HTTP benchmarking tool. https://github.com/wg/wrk.
[9] P. H. Hargrove and J. C. Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series* (sep 2006), 494–499.
[10] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 380–394.
[11] Hex-Rays. 2020. The IDA Pro Disassembler and Debugger. https://www.hex-rays.com/products/ida/.
[12] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Symposium on Security and Privacy (SP)*. IEEE, 618–635.
[13] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. 435–445.
[14] Y. Jiang, D. Wu, and P. Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, USA, 12–21.
[15] Y. Jiang, C. Zhang, D. Wu, and P. Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *Proceedings of the International Symposium on High Assurance Systems Engineering (HASE)*. ACM, 122–131.
[16] S. H. Kan. 2002. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional.
[17] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (London, UK). 121–132.
[18] H. Koo, S. Ghavamnia, and M. Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the European Workshop on Systems Security (EUROSEC)*. ACM, 9:1–9:6.
[19] J. Landsborough, S. Harding, and S. Fugate. 2015. Removing the Kitchen Sink from Software. In *Companion Publication of the Annual Conference on Genetic and Evolutionary Computation*. ACM, USA, 833–838.
[20] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200.
[21] G. Misherghi and Z. Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 142–151.
[22] Nathan P Myhrvold. 1997. The Next Fifty Years of Software. http://hartenstein.de/EIS2/next50years.pdf.
[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. *SIGOPS Oper. Syst. Rev.* 36 (Dec 2002), 361–376.
[24] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. 2022. PacJam: Securing Dependencies Continuously via Package-Oriented Debloating. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*. 903–916.
[25] C. Porter, G. Mururu, P. Barua, and S. Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 164–180.
[26] Georgios Portokalidis and Angelos D. Keromytis. 2011. REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points. In *Proceedings of the International Workshop on Security (IWSEC)* (Tokyo, Japan). Springer-Verlag, Berlin, Heidelberg, 16–32.
[27] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium*. USENIX Association, Santa Clara, CA, 1733–1750.
[28] A. Quach, A. Prakash, and L. Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the USENIX Security Symposium*. USENIX Association, USA, 869–886.
[29] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). 482–501.
[30] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 335–346.
[31] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. 2005. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *The International Journal of High Performance Computing Applications* (2005), 479–493.
[32] SecurityFocus. 2013. Nginx 'ngx_http_parse.c' Stack Buffer Overflow Vulnerability. https://www.securityfocus.com/bid/59699.
[33] Selectel. 2014. ftpbench. https://github.com/selectel/ftpbench.
[34] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 329–339.
[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[36] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. 2009. ASSURE: Automatic Software Self-Healing Using Rescue Points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 37–48.
[37] Standish Group. 2009. CHAOS report 2009. https://www.classes.cs.uchicago.edu/archive/2014/fall/51210-1/required.reading/Standish.Group.Chaos.2009.pdf.
[38] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. 2018. Perses: Syntax-guided Program Reduction. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 361–371.
[39] Synopsys, Inc. 2020. The Heartbleed Bug. https://heartbleed.com/.
[40] WinSCP. 2020. Supported File Transfer Protocols. https://winscp.net/eng/docs/protocols.
[41] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* (Feb. 2002), 183–200.
[42] H. Zhen and T. Gang. 2019. Rapid Vulnerability Mitigation with Security Workarounds. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*. ISOC, Reston, VA, USA.
[43] T. Zimmermann, N. Nagappan, and L. Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 421–428.

A Examples from Razor Analysis

This section contains examples of the analysis we performed on the coreutils programs debloated by Razor. We show two types of failures: failures corresponding to over-debloating (i.e., wanted functionality dropped) ; failures corresponding to under-debloating (i.e., unwanted functionality preserved).

A.1 Required Functionality Dropped

We identify these cases by simply running the RAZOR benchmarks provided by the authors. Some of the functionalities and command line fails in the prepared tests although they where used in the training cases.

chown-8.2 Fails on recursive mode if used on non-empty directories.

```
./chown.org.debloat -R  root:root  d1/d1/d1/d1
```

chown-8.2 Fails if multiple files used as input.

```
./chown.org.debloat root:root  file1 file2
```

rm-8.4 Fails in recursive mode if used on non-empty directories.

```
./rm.org.debloat -rf root:root  d1
```

tar-1.13 Fails on one of the input files from the Razor test examples

```
./tar.org.debloat cf tmp.tar obj.bz2
```

A.2 Unwanted Functionality Included

We identify functionalities preserved in the debloated binary although they where not used in the training cases. To identify these we explore the debloated program by testing manually different untrained features.

date-8.21 some of the options for formatting the output execute even though not present in the training test cases (note that some of the untrained options are debloated).

```
./date.orig.debloated -d "1995-1-17" +%a
./date.orig.debloated -d "1995-1-17" +%b
./date.orig.debloated -d "1995-1-17" +%C
./date.orig.debloated -d "1995-1-17" +%e
./date.orig.debloated -d "1995-1-17" +%g
./date.orig.debloated -d "1995-1-17" +%n
./date.orig.debloated -d "1995-1-17" +%N
./date.orig.debloated -d "1995-1-17" +%z
./date.orig.debloated -d "1995-1-17" +%:z
./date.orig.debloated -d "1995-1-17" +%Z
```

grep-2.19 the option of printing the context of the regex match (`-NUM`) executes even though not present in the training test cases (note that options `-CNUM` and `-context=NUM` which are alternatives of `-NUM` are debloated).

```
./grep.orig.debloated -1 [0-9] ../test2
```

mkdir-5.2.1 the verbosity option executes normally even though not present in the training test cases.

```
./mkdir.orig.debloated -v -p d1/d2/d3
```

uniq-8.16 the options `-zero-terminated` and `-all-repeated` execute normally even though not present in the training test cases.

```
./uniq.orig.debloated --all-repeated=prepend file
./uniq.orig.debloated --all-repeated=separate file
./uniq.orig.debloated --zero-terminated file
```

B Analysis of *FS-edges*

We analyse the detected *FS-edge* for our evaluated application shown in Table 4. We use blue and yellow to highlight the lines of code corresponding to the source and destination of the *FS-edge* respectively.

IMAGEMAGICK-file: In the test of features supporting TIFF / SVG / PNG / JPEG, we detected the same, proper *FS-edge* with any combination of 2-images and *M*-mutations. The *FS-edge*, as shown in Listing 1, is an indirect call to register the module responsible for the target format. Removing the *FS-edge* prevents registration of the module and thus, indeed disables the target format. Moreover, each module is designated for the target format and therefore, removing the *FS-edge* does not hurt other formats.

IMAGEMAGICK-UI: We detected the correct *FS-edge* for every feature, using any combination of 2-images and *M*-mutations. Listing 2 shows the *FS-edges* for the UI features we disabled. It is an indirect jump from the switch checking the UI click to the case implementing the corresponding action. Cutting off the edge disables the feature without affecting the others.

```
MagickBooleanType RegisterStaticModule(...) {
      ...
      if (MagickModules[i].registered == MagickFalse)
            /* Indirect call to TIFF module */
            (void)(MagickModules[i].register_module)();
      ...
}
ModuleExport size_t RegisterTIFFImage(void) { ... }
```

Listing 1: The Branch handling different image types in IMAGEMAGICK.

```
/* FS-edge is an indirect jump to a case in switch*/
   static Image *XMagickCommand(...}
 switch (command) {
      case CropCommand: { // Crop image.
  (void) XCropImage(display,resource_info,
  windows,*image,CropMode,exception);
  break;
      }
      case ChopCommand:  {...} // Chop image.
      case FlopCommand:  {...}  // Flop image scanlines.
      case FlipCommand:  {...}  // Flip image scanlines.
      case RotateRightCommand:  {...} // Rotate image
      case ShearCommand:  {...}
      ...
 }
```

Listing 2: *FS-edges* in IMAGEMAGICK for the UI features.

EVINCE: For every feature in EVINCE, we detected the correct *FS-edge* with any combination of 2-files and *M*-mutations. The *FS-edge* is an indirect call to the function implementing the feature. The related code is in Listing 3.

```
/*The source basic block is in the gnome library. The feature-specific
      edge is a callback to the function ev_window_cmd_file_print.*/
GActionEntry actions[] = {
   { "print", ev_window_cmd_file_print },
    ...%
   };
   ...
   /* A gnome function registering the callback functions*/
   g_action_map_add_action_entries (ev_window, actions);
   ...
   /* The callback function called by the gnome framework */
   static void ev_window_cmd_file_print (...) { ... }
```

Listing 3: *FS-edge* in Evince for Print feature.

NGINX: By using any 2-urls and *M*-mutations, we detected the correct *FS-edge* for each feature supporting a HTTP request method. The *FS-edge*, following the same pattern shown in Fig. 3, checks the method and picks the proper handler. With this edge cut off, the handler is no longer accessible and the method is disabled. We also detected an *FS-edge*, shown in Listing 4, to disable the "Chunked Encoding " feature without hurting other functionality (only one combination of 2-mutations is available in this case). The *FS-edge* is a jump from a check of the size of the method name to a check of the actual method name. By intuition, the *FS-edge* can be unsafe since other method names may share the same length. Fortunately, this did not happen because "Chunked" is the only method name with size of 7.

```c
/* FS-edge is a jump from a check of the length of method name to a
       check of the actual method name. "Chunked" is the only method
       with size of 7.*/
ngx_int_t ngx_http_process_request_header(...){
      ...
      if ( r->headers_in.transfer_encoding->value.len == 7 &&
       ngx_strncasecmp(r->headers_in.transfer_encoding
       ->value.data,(u_char *) "chunked", 7) == 0 ){
         r->headers_in.content_length = NULL;
         r->headers_in.content_length_n = -1;
         r->headers_in.chunked = 1;
         }
      ...
}
```

Listing 4: *FS-edge* in NGINX for the "Chunk-Encoding" feature.

```c
int ngx_http_discard_request_body_filter(...){①
    if (r->headers_in.chunked) {
       rc = ngx_http_parse_chunked(r, b, rb->chunked);
       ...
    }
}
int ngx_http_request_body_filter(...){
    if (r->headers_in.chunked) {
      return ngx_http_request_body_chunked_filter(r, in);②
    }
    ...
}
int ngx_http_proxy_input_filter_init(...){
    if (u->headers_in.chunked) {
      u->pipe->input_filter =
          ngx_http_proxy_chunked_filter;③
      u->input_filter = ngx_http_proxy_non_buffered_chunked_filter;④
      ...
    }
}
```

Listing 5: Control flow to reach CVE-2013-2028-related code.

ZIP: The tests with ZIP We detected the correct *FS-edge* for every feature, using any combination of 2-files and *M*-mutations. Listing 6 shows the *FS-edge* of an indirect jump from the switch checking the command line to the case implementing the target feature.

```c
/* The feature-specific edge is a jump from a switch to the case "O_TT"
      */
int main(argc, argv)
{
   ...
   switch (option)
   {
      ...
      case o_TT :  /* command path to use instead of 'unzip -t ' */
      if (unzip_path)
         free(unzip_path);
      unzip_path = value;
      break;
      ...
   }
}
```

Listing 6: *FS-edge* in `Zip` that corresponds to the "Archive test command" feature. (Conditional jump)

EXIM: We detected the *FS-edge* shown in Listing 7 for the "startup script" feature in EXIM, using any combination of *M*-mutations. The *FS-edge* is a conditional jump to the code launching the startup script. With this jump disabled, the "startup script" can no longer work but no other functionality is affected.

```c
/* FS-edge is a conditional jump to the code launching the startup
       script. */
int main(int argc, char **cargv){
 ...
   if (opt_perl_at_start &&  opt_perl_startup != NULL ){
   errstr = init_perl(opt_perl_startup);
  .../*code handling Chunked Encoding''*/
 }
```

Listing 7: *FS-edge* in EXIM for the "startup script" feature.

BASH: We detected the *FS-edge* shown in Listing 8 for the "defining functions by environment variable" feature in BASH, with any 2-commands and *M*-mutations. The *FS-edge* is a conditional jump to the code reading and executing the functions defined in the environment variable. Cutting off the *FS-edge* will prevents execution of those function without harm to any other functionality.

```c
/* FS-edge is a jump to the code reading and executing the function
       defined in the environment variable"*/
void initialize_shell_variables (...){
   ...
 if (privmode == 0 && read_but_dont_execute == 0 &&
         STREQN ("() {", string, 4) ){
       string_length = strlen (string) ;
       ...
   }
}
```

Listing 8: *FS-edge* in BASH for "functions in env. variables" feature.

PROFTPD: We detected the *FS-edge* shown in Listing 9 for "CPFR" feature in PROFTPD with $M = 3$ mutations. The *FS-edge* is a conditional jump to the code executing the "CPRF" command after a string compare with the user input. By disabling this edge, the user command "CPFR" cannot be treated without affecting other functionalities.

```c
      /* FS-edge is a jump to the code that implements the "CPFR"
            command */
      MODRET copy_cpfr(cmd_rec *cmd) {
         ...
         if (cmd->argc < 3 ||
            strncasecmp(cmd->argv[1], "CPFR", 5) != 0) {
              return PR_DECLINED(cmd);
         }
         CHECK_CMD_MIN_ARGS(cmd, 3) ;
         .../* code implementing "CPFR".*/
```

Listing 9: *FS-edge* in `ProFTPD` for the CPFR feature (*conditional jump*).

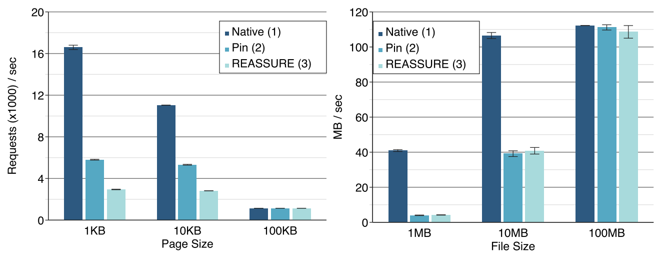
Table 6: Rescue points generated by F-blocker. For RP distance (c)/(b)/(a): The values correspond to (a) the depth of the function containing the *FS-edge*, (b) and the depth of the nearest rescue point function in the call trace (c) the depth of the detected Rescue point function in the call trace (distance from `__libc_start_main()`).

Feature	RP Distance	Detection Method	Error Returned	RP Function Name
ImageMagick (file)	6/7/8	Pointer return	0 (NULL)	`GetMagickInfo`
ImageMagick (UI)	3/4/5	Syscall failing	0 (old value = 1)	`DisplayImageCommand`
Evince	23/23/28	Pointer return	0 (NULL)	`g_action_group_activate_action`
Exiv2	4/4/4	Pointer return	0 (NULL)	`Action::Insert::clone_()`
Nginx	7/9/10*	Syscall failing	-1 (old value = 0)	`ngx_epoll_process_events`
ProFTPD	8/8/8	Pointer return	0 (NULL)	`copy_cpfr`
BusyBox	6/6/6	Syscall failing	1 (old value = 0)	`wget_main`
Exim	1/1/1	Syscall failing	1 (old value = 0)	`main`
Bash	1/1/3	Syscall failing	1 (old value = 0)	`main`
Zip	1/1/1	Syscall failing	1 (old value = 0)	`main`

* Some of Nginx functions uses function parameters to pass errors.

C Overhead of Deploying RPs with REASSURE

We measured the overhead of REASSURE when deploying our RPs using Nginx and ProFTPD to show that our RPs are not more heavyweight than those proposed by prior works. Note that REASSURE can incur significant overheads over native execution, because it builds on Pin, however, it is ideal for fast prototyping. In production environments, more efficient checkpoint-rollback systems should be used [2, 9, 23, 31, 36].



(a) Nginx (b) ProFTPD

Figure 6: Performance of Nginx and ProFTPD with and without REASSURE. (a) We used WRK [8] to measure requests / second with different page sizes. (b) We used `ftpbench` [33] to measure the throughput with different files sizes.

We ran Nginx and ProFTPD on a 2-core Xeon E3-1270 V2 @ 3.50GHz Xen VM with 29GB of RAM (Debian 4.9.168-1+deb9u5, Xen 4.1). The benchmark clients ran on another host with a 4-core Xeon E3-1270 v6 @ 3.80GHz and 64GB of RAM (Ubuntu 16.04.6 LTS), connected over 1Gb/s Ethernet to the server. The client opens 10 simultaneous connections and sends requests for 1 minute with random files of different size (1KB, 10KB, and 100KB GET HTTP requests for Nginx and 1MB, 10MB, and 100MB RETR FTP requests for ProFTPD). We used 2 threads for the Nginx client to saturate the server. For comparison, we considered three different scenarios: (1) running the application natively; (2) running the application with Pin; (3) running the application with REASSSURE. The experiments were repeated five times, and we show the mean and standard deviation (SD) in Figures 6a and 6b.

In the Nginx evaluation with requests of small files (1KB), REASSURE incurs x5.6 and x1.98 overhead, respectively comparing to native execution and Pin. The overhead is because the rescue point sits on a critical path, which is activated in nearly every request. When the file size increases to 100KB, we observed no significant overhead. This is potentially because the bottleneck moves from the CPU to the network and the frequency of requests is lower (but they take longer). In the case of Nginx, REASSURE added no observable overhead over Pin. The reason is that the rescue point is not activated during the file transfer requests issued by the benchmark, representing the best scenario.

To sum up, the overhead incurred by REASSURE depends on the unwanted features and the correlation between the unwanted features and other features. Even if significantly faster checkpoint-restart is used, rescue points on the critical path of the server are bound to incur some overhead. However, in many cases unwanted features are in rarely executed code and overhead will be minimal.