Disabling Unwanted Functionality in Binary Programs

Mohamad Mansouri
Thales SIX GTS / EURECOM

Jun Xu
University of Utah

Georgios Portokalidis
Stevens Institute of Technology

***Abstract*—Driven by the diversification of application scenarios and the increase in market needs, software systems are rapidly integrating new utilities and functionalities, usually presented as new *features*. This frequently results in the problem of feature creep: available features exceed the desired functions of users. It also affects software attack surfaces, as more code carries the risk of more vulnerabilities. To mitigate these security concerns in binary programs, we propose F-DETECTOR and F-BLOCKER. F-DETECTOR is able to detect and disable features activated by most common types of inputs (command lines, file/network data, graphical interfaces, and configuration options). Given a small set of inputs that activate an unwanted feature and inputs (derived by the former) that do not, F-DETECTOR combines dynamic tracing and static analysis to detect the feature entrance — a branch on the control flow graph that is only traversed when the unwanted feature is activated. F-BLOCKER uses the discovered feature entrance to disable features without affecting 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 F-DETECTOR and F-BLOCKER for the Linux platform and evaluate it with 192 features (corresponding to 10 known vulnerabilities) from 19 programs. Results show that they can detect and disable all features with few errors, affecting only one of the tested applications, while they outperform previous works.**

I. 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 never use a considerable part of available functionality [34]. We can view the set on unused 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 [39], [17] and serious vulnerabilities [1] have been discovered in rarely used features.

Eliminating code bloat can, thus, improve security because it reduces the attack surface of applications by eliminating code that may contain known and unknown vulnerabilities. Debloating applications can be done in an aggressive, greedy manner by eliminating *all unused* features. Many previous works [12], [32], [26], [15], [18], [8], [35], [28], [6], follow this approach. These works use test cases covering necessary functionality to identify and remove code not needed, which usually corresponds to a large number of features. The danger is that they tend to remove required code that was not covered by the test cases, such as environment, configuration, and error-handling code.

Debloating can also be done by only eliminating specific *unwanted* features, which has a lower chance of erroneously removing required code. One strategy of doing so is to explore using execution traces of test cases to remove the unwanted feature, as demonstrated by Landsborough et al. [19]. However, their approach relies on the availability of complete test suites of the unwanted feature and thus, are limited to small, simple utilities (e.g., `sha1sum`). An alternative strategy is to rely on users identifying seed functions uniquely tied to the unwanted feature and run dynamic/static analysis to either remove code connected with the seed functions [16] or block execution flowing into them [5]. This line of efforts require prior knowledge about the seed functions responsible for implementing the unwanted feature, incurring high manual efforts and cannot be applied by non-developers. Moreover, regardless of which strategy we consider, the existing solutions resort to terminating the application when the unwanted feature is activated, failing to support the continuity of normal service and avoid the potential damage.

In this paper, we present a system for disabling *unwanted* features in binary applications without carrying the limitations of existing solutions. The system runs two major components, F-DETECTOR and F-BLOCKER, to achieve its goal.

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. F-DETECTOR operates in a semi-automatic way. 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. It combines the differences observed in the traces with information obtained through static analysis of the application to determine a control-flow branch that dominates the feature as its entrance.

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 decide a function that can work as a *rescue point* or precisely, a location where execution can rollback and an error can be raised to activate built-in error-handling. F-BLOCKER's runtime component then instruments the application to deploy the rescue point. Once the feature entrance is hit, F-BLOCKER rollbacks the execution to the rescue point and triggers the built-in error handling to disable the unwanted feature safely.

While our work is not the first to target feature removal, it brings several unique, widely desired advantages. First, it only

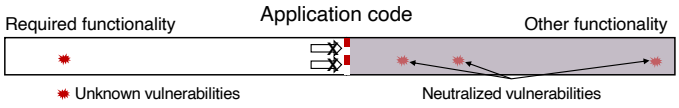


Fig. 1. Eliminating vulnerabilities through debloating.

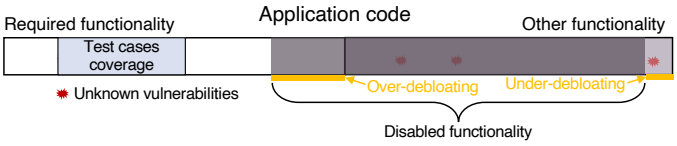


Fig. 2. Debloating based on retaining wanted functionality.

```
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 ...
```

Execution flow
GET method
PUT method
GET & PUT common
Others

Fig. 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.

requires a few inputs and some basic understanding of target-software features to minimally mutate them, decreasing the burden placed upon and required expertise of users. Second, it only disables a single control flow edge. Identifying this edge is more tractable in comparison to finding all the code blocks corresponding to a feature, also significantly reducing the potential side effects to other functionality. Third, it is designed in a way not limited to a specific type of program or feature. It can handle 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 because crashes can cause data loss, beyond inconvenience.

We have implemented prototypes of F-DETECTOR and F-BLOCKER, which have been evaluated using 192 features from 19 applications (including command-line utilities, servers, and GUI applications). To our knowledge, this is the most extensive experimental evaluation of a system removing unwanted features. We manually verified the evaluation results. We found that our system is able to detect the correct feature entrance for most of the tested features, regardless of the inputs and mutations. Only when handling a small set number of features, it presented errors because low-quality inputs or mutations are selected. In addition, our system disables 10 known vulnerabilities rooted within the tested features.

In summary, we make the following contributions:

- We design and implement a system that is able to semi-automatically identify all types of program-features in stripped binaries and disable them without affecting program availability.
- 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 code.
- 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.
- We evaluate our system using 19 applications and 192 features with 10 associated vulnerabilities (CVEs). Our results show that it can disable all features and insulate the application from the vulnerabilities.

II. Background and Motivation

A. Reducing Attack Surface through Debloating

Large and complex software is more likely to contain bugs and vulnerabilities [39], [17], hence, disabling or removing the usually numerous but rarely used features [34] reduces its attack surface. By removing or ensuring undesired functionality is unreachable during execution, we neutralize any vulnerabilities rooted within it, as illustrated in Fig. 1. This is a proactive measure that eliminates known, but more importantly unknown (i.e., zero-day) program vulnerabilities.

Eliminating unwanted functionality, or *features*, can be often done during compilation, through configuration options made available by the developers. For example, Debian GNU/Linux offers various versions of the popular VIM editor, with *vim-tiny* including only about twelve features out of more than a hundred in the full version. However, some applications may not include options for disabling all unwanted features or source code may not be available (e.g., proprietary software). To address these issues, recent research has focused on automatically debloating software by removing or disabling unwanted functionality. The produced approaches can be classified into two categories: the ones that aim to identify required functionality and eliminate *all* other functionality, and the ones that aim to disable specific unwanted features, one at a time. We discuss them below.

B. Different Debloating Strategies

1) Retaining Wanted Functionality: Works in this direction [12], [32], [26], [15], [18], [8], [35], [28], [6] are based on profiling applications using test cases, or *inputs*, corresponding to required functionality. Based on the code covered when executing with these training inputs, they estimate the greater code area related to the required functionality. Code corresponding to other functionality is erased or disabled. This strategy can potentially maximize the amount of code and vulnerabilities removed.

TABLE I. EVALUATING RAZOR WITH COREUTILS. ● INDICATES WE DISCOVERED A PROBLEM AND ○ THAT WE DID NOT.

Application	Debloating	
	Over	Under
`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`	○	●

2) Disabling Unwanted Functionality: Debloating can also be done by eliminating specific *undesired* features. One way to achieve this is to rely on users identifying seed functions, key to the targeted feature, and using dynamic and static analysis to either remove all code associated to the feature [16] or block execution flowing into it [5]. Another way is to remove features using run-time profiling. For instance, Landsborough et al. [19] collect instruction traces with test cases for both wanted and unwanted features. They identify code that was activated during runs with the unwanted-feature inputs but not with wanted-feature inputs. By overwriting these code segments with no-op instructions (`nop`), the corresponding functionality is disabled.

C. Retaining Functionality v.s. Disabling Functionality

Functionality debloating is hard to do because it often incurs two problems:

Over-debloating leads to erroneously disabling code associated with required functionality, effectively *breaking* applications when debloated code paths are activated.

Under-debloating leaves some unwanted functionality in the program. This is seemingly more innocent, however, it may introduce a *false sense of security*. For instance, when a new vulnerability is discovered, affecting a feature assumed to have been completely disabled. Unless users explicitly test for a feature, under-debloating may have left a now known vulnerability in their application.

Retaining wanted functionality has been facing challenges to mitigate the two problems above. Reviewing specific approaches, CHISEL [12], a notable recent work, takes inputs that offer broad coverage of wanted functionality and leverages reinforcement learning to estimate required code. Follow-up studies [26] show that CHISEL is prone to over-debloating and, in extreme cases, it can even *introduce vulnerabilities* in programs due to removing checks. More recently, RAZOR [26] defines a set of heuristics for expanding the code paths exercised by inputs corresponding to desired functionality, aiming to detect the additional code needed. While less aggressive than CHISEL, our experiments with the prototype made publicly available by the authors confirmed that RAZOR can result in both over- and under-debloating. Table I summarizes the findings of our experiment with RAZOR, where the training and testing inputs from the original paper are used to detect over-debloating and new testing inputs for unwanted features are included to identify under-debloating.

Underneath the difficulties for retaining wanted functionality to avoid over- or under-debloating, there are some fundamental reasons. Driven by the principle of only keeping the wanted functionality, this strategy has to **remove as much unneeded code as possible**. Without perfect inputs to cover all the code needed by the desired functionality (which is typically the case in practice), removing a large amount of code leads to a higher chance of errors (i.e., over-debloating).

In contrast, disabling unwanted functionality focuses on removing specific functionality. Principally, it only has to **trim the minimal code mandated by the unwanted functionality**. In the general sense, removing less code reduces the chance of errors and thus, disabling unwanted functionality is a better strategy to avoid over-debloating. On the other side of this argument, removing less code in general indicates a higher chance of retaining the unwanted functionality. Hence, disabling unwanted functionality can be more inclined to under-debloating. Fortunately, recent literature [16], [5] has unveiled that a functionality is often uniquely tied to a small piece of code (e.g., a key function), and preventing execution to such code can validly disable the functionality. That is, coupling with approaches to detecting the feature-associated code, disabling unwanted functionality can reduce under debloating.

To sum up, disabling unwanted functionality is better principled to avoid over- or under-debloating, which motivates us to follow this strategy to perform debloating.

D. Limitations of Existing Solutions to Disable Functionality

We are not the first attempting to disable unwanted functionality. However, existing solutions [16], [5] in the area are fundamentally limited in two aspects.

First, they rely on manual annotation or understanding (at least parts of) the implementation to detect feature-associated code constructs, which is highly complex or even impossible on binaries and cannot be applied by non-developers. Second, they tend stop unwanted functionality by simply terminating the execution (e.g., replacing all unwanted code with an invalid instruction [19]). This makes them impractical for servers or applications where data loss will occur when an unwanted feature is used (e.g., image editing applications).

In this paper, we aim to provide a novel functionality-removing solution to address the above two limitations.

III. DESIGN OVERVIEW

A. Key Insight and High-level Idea

We aim to disable unwanted functionality in binaries. Our key insight is that the functionality in programs, assuming it does not always execute independently of the input, is often activated or controlled by a control flow branch.

Conditional branches are often used to set up state variables to control the desired functionality. Fig. 3 shows an example. The conditional branch from line 8 to 9 is only executed for HTTP requests with the PUT method. The request's

```
1 MagickBooleanType RegisterStaticModule(...) {
2       ...
3       if (MagickModules[i].registered == MagickFalse)
4             /* An indirect call to the TIFF module is made using
                    index i on dispatch table MagickModules */
5             (void)(MagickModules[i].register_module)();
6       ...
7 }
8 ModuleExport size_t RegisterTIFFImage(void) { ... }
9 ModuleExport size_t ... /* other modules: PNG, JPEG, etc. */
```

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

state update on line 9 and then leads to the execution of PUT-related functionality in NGINX. Therefore, disabling this edge (e.g., by redirecting it to an aborting instruction) would disable the support of PUT-method in NGINX without having to identify all code blocks used in its implementation.

Indirect branches, like indirect calls, are another popular way to activate functionality. Typically, code before the indirect branch sets the target to point to code implementing the desired functionality. Once the branch is hit, the functionality will start executing. Listing 1 presents such a case. To process an image in the IMAGEMAGICK viewer and editor, the appropriate module is invoked through an indirect call to a different function depending on the image's type. For instance, the call on line 5 will only target `RegisterTIFFImage()` for TIFF images, and disallowing it will disable TIFF-related functionality.

We exploit the above insight to disable unwanted functionality or *features* ($\mathcal{F}$). The idea is to identify the first control-flow edge, which we call *feature-specific edge* (*FS-edge*), that controls an unwanted feature and block this edge. Similarly to prior work, we focus on disabling functionality that does not always execute, but instead, its activation depends on the *inputs* provided to the program. To clarify our scope, we consider any data that can be used by the program as inputs. Specifically, we consider the following inputs **whose value represents different features**: command-line options, network-protocol and file-format fields, configuration variables stored in files, shell environment variables, and clicking on graphical user-interface (GUI) elements. In the rest of this paper, we denote inputs that lead to the activation of an unwanted feature $\mathcal{F}$ as $I_{\mathcal{F}}$ and other inputs as $\neg I_{\mathcal{F}}$.

B. Disabling Unwanted Features with F-DETECTOR

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. Its design follows Fig. 4, which we highlight below.

1) Preparing Test Cases: Similarly to the approaches discussed in §II-B2, 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 caution us, randomly selecting test cases can lead to problems. Throughout empirical 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 popular types of inputs that F-DETECTOR handles, summarized in Table II.

Moreover, we use this minimal-mutation process to produce multiple different sets of $\neg I_{\mathcal{F}}$, allowing us to apply our *FS-edge*-detection multiple times. Execution traces with different $\neg I_{\mathcal{F}}$ can potentially produce different data, which further strengthens *FS-edge* detection and avoids debloating errors.

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

❶ we start by filtering out edges, keeping only edges taken by all the test cases in $I_{\mathcal{F}}$ but never by test cases in $\neg I_{\mathcal{F}}$. Otherwise, either $\mathcal{F}$ remains alive or other features will be affected;

❷ we eliminate edges that come from a utility function (e.g., `strcmp` from `libc`) because such functions intend to support a large variety of features;

❸ if a remaining edge corresponds to a conditional branch (`cbr`) and the code at the destination can only be reached through this edge, we consider the edge a candidate *FS-edge*. Otherwise, we discard the edge. The goal is to pick `cbr` that *uniquely* controls execution of its destination, thus offering a better probability to fully disable the unwanted feature. Similar heuristics apply for other types of edges (§IV);

❹ we pick the candidate *FS-edge* that occurs earliest in the trace. If the *FS-edge* is chained with other `cbr`-based *FS-edge* candidates, we consider the deepest one in the chain as the final *FS-edge*. Otherwise, we simply pick the earliest *FS-edge*. Prioritizing the earliest chain of *FS-edge* candidates helps block more feature-related code. Considering the deepest one in a chain is to accommodate complex condition checks where the shallower checks control feature groups and the deeper ones control individual features (e.g., the example in Fig. 3 first checks for methods of the same length and then the specific method). The benefit of doing so is a reduction of impact on other 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:

$$\begin{aligned} \text{PUT:} &\quad 1 \rightarrow 3, 4 \rightarrow 8 \rightarrow 9, 10 \rightarrow 14 \\ \text{GET:} &\quad 1 \rightarrow 3, 4 \rightarrow 5, 6 \rightarrow 14 \end{aligned}$$

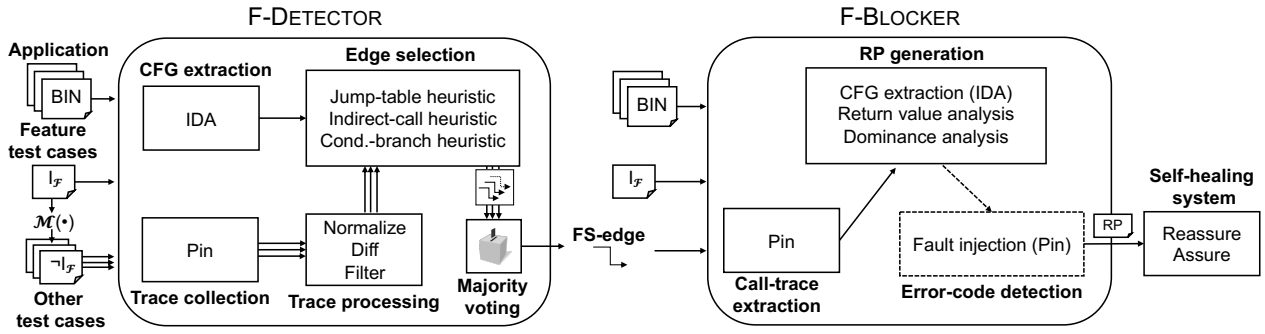


Fig. 4. Approach overview. Given a set of test cases, including both $I_{\mathcal{F}}$ and $\neg I_{\mathcal{F}}$ inputs, F-DETECTOR attempts to detect the control-flow edge (*FS-edge*) responsible for activating a targeted unwanted feature $\mathcal{F}$. F-BLOCKER generates a rescue points (RP) for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

Our algorithm will exclude $1 \rightarrow 3, 4$, following ❶, and $9, 10 \rightarrow 14$, following ❸. Edge $3, 4 \rightarrow 8$ is *initially* picked because of it is the earliest in the trace. Finally, we decide that the *FS-edge* is $8 \rightarrow 9$, as it is chained after $3, 4 \rightarrow 8$. 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. $1 \rightarrow 3, 4$ would be initially picked due to ❷-❹. Finally, ❹ would pick edge $8 \rightarrow 9$ because it is the last edge in a chain of valid conditional branches.

3) 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 through $\mathcal{M}()$. 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 §II. However, the *FS-edge* candidates could still be used to guide analysts and help them discover the correct *FS-edge* manually.

C. Disabling $\mathcal{F}$ and Exploring Survivability with F-BLOCKER

Given an *FS-edge*, we can easily disable $\mathcal{F}$ by terminating the application when the edge is traversed. We can accomplish this by overwriting the *FS-edge*'s destination, which is also the only edge leading to it, with a single-byte instruction like `int3`. This approach of disabling $\mathcal{F}$ is more robust than prior works that overwrite large "swathes" of code, since we literally only modify a single byte. However, this is still undesirable for servers and applications where data-loss may occur because of an uncontrolled exit.

How can we provide continuity of service when the disabled functionality is triggered? Our insight is that we can treat disabled functionality activation, as an unanticipated fatal error. We can then leverage techniques introduced by works on software self-healing to recover from these errors [33], [25]. Specifically, these systems introduce *rescue points* (RP) to enable software to handle unanticipated faults. RPs are functions returning error code(s) that the application already handles. When an unknown error occurs, execution is restored to an appropriate RP and a valid error code is returned, recovering execution by virtualizing and repurposing existing error-handling code.

How do we leverage rescue points? 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 (or snapshot) of the process or system state is created by the self-healing system. Traversing the *FS-edge* ($8 \rightarrow 9$) will trigger a fault, which in turn will cause 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 handle the error and keep operating. If the *FS-edge* is not hit, the checkpoint state is released upon return of the RP function.

To leverage software self-healing, we introduce F-BLOCKER, a system that automatically attempts to define a rescue point that can be used to recover from the injected error raised by a disabled feature. F-BLOCKER relies on dynamic and static analyses to find a function that can serve as a rescue point. The generated RP can be used with existing systems, such ASSURE [33] and REASSURE [25], for self-healing.

IV. F-DETECTOR

Fig. 4 depicts a high-level overview of F-DETECTOR. To disable $\mathcal{F}$, 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 $\mathcal{F}$ (i.e., $\mathcal{M}(I_{\mathcal{F}}) = \{\neg I_{\mathcal{F}_1}..\neg I_{\mathcal{F}_n}\}$). F-DETECTOR runs using $I_{\mathcal{F}}$ and each of the generated $\neg I_{\mathcal{F}}$ to determine multiple *FS-edge* candidates, one for each $\neg I_{\mathcal{F}}$. It then uses *majority voting* among the candidates to pick a single *FS-edge*.

The rest of this section describes the various components of F-DETECTOR in detail: §IV-A presents our guideline (rules of thumb) minimally mutating different types of $I_{\mathcal{F}}$ inputs; §IV-B describes how we trace and process the collected information to obtain an initial set of edges that include the *FS-edge*; finally, §IV-C describes our algorithm for identifying the candidate *FS-edge* using a group of heuristics.

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

F-DETECTOR requires both inputs triggering an unwanted feature ($I_{\mathcal{F}}$) and inputs that do not ($\neg I_{\mathcal{F}}$) for tracing. To

TABLE II. MINIMAL-MUTATION GUIDELINES SUMMARY FOR GENERATING $\neg I_{\mathcal{F}}$ BASED ON $I_{\mathcal{F}}$ AND INPUT TYPE.

$\mathcal{F}$ 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 $i \neq f$ }

prepare $I_{\mathcal{F}}$, users can just pick a (small) set of random test cases that activate the unwanted feature. To prepare $\neg I_{\mathcal{F}}$, we found that a practical way is *minimally mutating* $I_{\mathcal{F}}$ such that $\mathcal{F}$ is not activated. For instance, to disable the HTTP PUT method in the NGINX server (Fig. 3), users can prepare a single PUT request using utility like `curl`. They can then replace the method passed to the utility with other valid ones to produce $I_{\mathcal{F}}$, as shown below:

```
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 a set of guidelines on how popular input types handled by F-DETECTOR can be minimally mutated, by analyzing how the various applications we experimented with handle them. We summarize them in Table II and describe them in detail below:

Command-line Options Common in command-line programs, options are used to activate certain functionality of the program. Typically, a *parser* first processes them to update state accordingly (e.g., by asserting a variable). Linux programs commonly have both short and long version of options (e.g., `-R` is equivalent to `--recursive` in `chown`), so at least two test cases can be defined in $I_{\mathcal{F}}$. We can minimally mutate the targeted option by replacing it with other similar options (e.g., a long option with another long option) without modifying anything else. In applications with many command-line options, we can easily generate many different set of $\neg I_{\mathcal{F}}$.

Protocol Fields Many features in servers are activated based on the requests received. The server parses the request and activates some functionality based on the protocol field values in the request (e.g., the PUT method in HTTP). We should minimally mutate the (usually) single-input $I_{\mathcal{F}}$ by replacing the protocol field with other valid values, avoiding modifications to the common parts of the request (there can be differences mandated by the new field value). Multiple sets of $\neg I_{\mathcal{F}}$ can be generated based on alternative field values.

File Formats Applications that handle (various types of) files parse them and based on the file format (or specific fields in it), activate certain functionality. This is similar in many ways to protocol fields. For example, image viewer applications support multiple image file types and each of them could be considered as a separate feature. In this case, $I_{\mathcal{F}}$ contains one or more images of the unwanted format. We minimally mutate them by converting them to other formats (e.g., using a converter or the application itself). Depending on the image format and conversion capabilities, multiple test cases can be generated for $\neg I_{\mathcal{F}}$. We avoid randomly selecting the files in $I_{\mathcal{F}}$ and $\neg I_{\mathcal{F}}$, as there can be differences in metadata, or files may require additional functionality, unbeknown to us.

Configuration Variables Variables in configuration files can also control the use or not of a feature. For instance, when defined, the variable `perl_at_start` enables the Perl interpreter in the EXIM mailer to run the script assigned to the variable. In such cases, we can simply remove the given variable from the configuration to minimally change an $I_{\mathcal{F}}$ to $\neg I_{\mathcal{F}}$. If the variable can be assigned distinct values, we can instead replace the value assigned to the variable with other valid options.

Environment Variables Can be treated similarly to configuration variables.

GUI Actions In GUI applications, many features are triggered by a user action, delivered by a keystroke (e.g., shortcuts) or mouse click (e.g., clicking on a menu item). This causes the execution of a callback from the GUI framework being used, which will eventually execute the 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. To minimally mutate menu-item clicks, we can click on a different item under the same menu. For shortcuts, we can use a different keyboard shortcut. We can easily generate different sets of $\neg I_{\mathcal{F}}$ for most GUI applications, as they usually include numerous actions.

B. Execution-Trace Collection and Processing

F-DETECTOR collects execution traces of the application, which compromise the address of every executed basic block (BBL), with basic blocks being sequences of instructions that end with a control-transfer instruction. By collecting and comparing $I_{\mathcal{F}}$ traces against $\neg I_{\mathcal{F}}$ traces, we aim to identify a small set of control-flow edges, which will include the *FS-edge*. These edges will satisfy the following properties:

- They are present in *every* trace collected with $I_{\mathcal{F}}$.
- They are *never* present in traces collected with $\neg I_{\mathcal{F}}$.

1) Trace Normalization: Each trace may comprise multiple sub-traces, one for each thread of execution. Each sub-trace is identified by its thread ID tid, including all BBLs executed by the thread in the order they run. We first process the traces by normalizing them. For each sub-trace, we record the unique control-flow transitions performed by the thread, as a source–destination pair of BBLs (src-dst). We also include the position (pos) of the first appearance of each BBL and the number

of its appearances (num) in the sub-trace. All sub-traces are eventually merged into a single trace containing tuples in the form of $(src, dst, tid, pos, num)$, which correspond to unique edges across threads. tid, pos and num correspond to those of the thread sub-trace where they first appeared in. In the rest of this section, we will refer to normalized traces as *profiles*.

2) Profile Diffing: Next, we compare the collected profiles to obtain a first set of control-flow edges that will include the *FS-edge* we desire. We first generate set C by taking the intersection of all (src, dst) pairs in $I_{\mathcal{F}}$ profiles, which includes all the CFG edges that were taken consistently in all executions where the feature is activated. Second, we generate set E by taking the union of all edges in $\neg I_{\mathcal{F}}$ profiles. Finally, we subtract E from C to obtain a set without any edges appearing in $\neg I_{\mathcal{F}}$ profiles.

3) Utility-Function Filtering: Applications almost always use utility functions, like string-comparison functions, from libraries like `libc`. Their internal code typically does not uniquely relate to *any* $\mathcal{F}$, hence, the *FS-edge* will unlikely be located in them. Therefore, we exclude such edges by filtering out ranges that correspond to utility libraries, like `libc` or other user-configured libraries.

```
1  size_t CopyMagickString(char *dest, char *src, size_t length){
2    ...
3    for (n=length; n > 4; n-=4){
4      *q=(*p++);
5      if (*q == '\0') return((size_t) (p-source-1));
6      q++;
7      *q=(*p++);
8      if (*q == '\0') return((size_t) (p-source-1));
9      q++;
10   }
11   ...
12 }
```

Listing 2. A utility function from IMAGEMAGICK (V-7.0.9-5).

Applications may also include built-in utility functions. To eliminate them, we exploit the observation that they are called frequently even for basic workloads and discard edges that occur multiple times. For example, in IMAGEMAGICK function `CopyMagickString` (Listing 2) is called multiple times and contains a loop (causing internal edges to appear multiple times), which leads to exclusion.

C. FS-Edge Detection

To detect the *FS-edge* using the set of edges identified in the previous stage, we have devised a set of heuristics, based on how programs commonly "decide" to activate functionality. They allow us to overcome the limitations of working with execution traces obtained with a small number of inputs, which may include edges that are also associated with functionality other than $\mathcal{F}$. We start by grouping consecutively executed edges into *packs*. That is, each pack contains edges where the destination BBL of the first edge is the source BBL of the second one, and so on. We go over these packs in order, from earlier to later executed edges, searching for the earliest pack that contains an *FS-edge* according to our heuristics.

1) 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 indirect jumps, 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` that corresponds to a `switch` jump table is found, we treat it as a `cbr`. This heuristic targets applications that use a switch statement to conditionally activate $\mathcal{F}$, which was implemented by the compiler using a jump table. Our example in Fig. 3 contains one such `switch` (line 1) to check different method-string lengths. To determine if an `ijmp` is part of a `switch` instead of an indirect call, we utilize the IDA Pro disassembler [13] 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 that use a dispatch table containing pointers to functions associated with different features. Usually, an index value is used to obtain the appropriate pointer which is then called. Listing 1 shows one such use of a dispatcher table 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 frequently 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 ($1 \rightarrow 3$), followed by `if-then` statements testing for specific method names ($4 \rightarrow 8 \rightarrow 9$). 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, signifying that the chain of `cbrs` has ended;
- an edge to a BBL with multiple incoming edges occurs. This rule aims to exclude `cbr` 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 corresponds to a direct 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 enable a `cbr`-based *FS-edge* uniquely controls execution of its destination, increasing the chance of completely blocking the unwanted feature.

2) FS-Edge Detection in the Presence of Threads: Our algorithm expects that edges in each profile are ordered. However, absolute ordering in multi-threaded applications is very challenging, especially on multi-core architectures where threads execute in parallel. To address this issue, we apply our approach to the profile of each thread separately, which may identify one *FS-edge* for each thread of the application. We assume that the *FS-edge* that executed first caused the ensuing ones. To select it, we re-run the application with one of the inputs in $I_{\mathcal{F}}$, while we instrument it to record when *FS-edges* are traversed.

3) Majority Voting: F-DETECTOR applies the *FS-edge* detection algorithm multiple times using the $I_{\mathcal{F}}$ set and each of the $\neg I_{\mathcal{F}}$ sets. This results in multiple *FS-edge* candidates being initially generated. At this point, we have the following options: (i) be strict and only use the *FS-edge* if all candidate agree (unanimous decision), (ii) use majority voting and select the *FS-edge* (if any) that the majority of runs produced. In the evaluation, we use option (ii).

V. F-BLOCKER

F-BLOCKER automatically attempts to define a rescue point that can be used to recover from the injected and unexpected error which is *attempting to execute* $\mathcal{F}$. This rescue point can be used with an software self-healing system, such as ASSURE [33] or REASSURE [25]. The RP is a function that must satisfy the following criteria:

- **P1**: Every possible path leading to the *FS-edge* includes the function, so we can always "rescue" the application. For instance, the function containing the *FS-edge* always satisfies this criterion.
- **P2**: The functions returns an error code, which is handled by its callers. This is a key piece of error virtualization.
- **P3**: It is near the *FS-edge* to reduce overhead.

Fig. 4 presents an overview of F-BLOCKER's components. They are discussed in detail in the sections below.

A. Call-Trace Extraction

We start by running the application with the feature-triggering inputs that were used with F-DETECTOR ($I_{\mathcal{F}}$). In each run, we record several information including pairs of function callers-callees, function returns and potential return values (e.g., the value of register `RAX`), active memory mappings when returning, and system calls performed, along with their return values. Finally, when the *FS-edge* is hit, we record the call stack at that point and terminate.

B. Rescue-Point Generation

1) Dominance Analysis: To satisfy **P1**, the RP must be one of the functions in the call stack obtained in the first step. To determine which functions are eligible, we extract the CFG of the application to determine domination relations. Specifically, the functions that dominate the function containing the *FS-edge* (i.e., all execution paths go through them) are RP candidates. If a function in the call stack is address taken (AT), meaning the program contains a reference (pointer) to it, our analysis does not attempt to resolve all potential callers, as this is impossible to do accurately. Instead, we rely on call-trace data to assign the callers (usually one) based on what was observed during tracing. This may be an underestimate, however, if one of the callers of an AT function is finally selected as an RP, manual analysis can be employed to try and determine if **P1** is indeed satisfied before deploying.

```
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(...)
RP             └→ ngx_int_t ngx_http_parse_request_line(...) {
                     ... /* error retval = { ?? } */
                     int3 r->method = NGX_HTTP_PUT; /* unwanted feature */
                     ...
```

Fig. 5. Rescue Point for Disabling NGINX's PUT method.

2) Return-Value Analysis: For each of the eligible functions, we try to automatically determine: (i) if they return a value (e.g., not a `void` return type), (ii) which values correspond to errors, and (iii) if they are handled by the application. Functions that satisfy these criteria also satisfy **P2**. Binary applications return values according to the calling conventions used, which in x86 architecture, it is commonly done through the `EAX/RAX` registers. We statically analyze applications to find how these registers are used to establish (i) and (iii), as follows:

- For functions that are not address-taken, we analyze all their callers to determine whether `EAX/RAX` is used, right after a call returns, without being set first. This indicates they use a value set/returned by the function.
- 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 (reading) its value before returning. This indicates that the function is always returning a value.

For establishing returns values that signal errors, namely (ii), previous research has explored static analysis based approaches [14], [38]. These works, however, often involve inferences that may undermine the safety of our system. Instead, we adopt a *dynamic* approach inspired by two observations: functions returning pointers often return `NULL` to indicate an error and functions issuing system calls often test for and return an error code to their callers. Based on these and the values observed during tracing, we define the following two rules:

- if the return value lies in the address range of mapped memory, we consider it to be a pointer and a valid error code is `NULL`.
- otherwise, we assume the function returns integers and we use the method described below to determine error codes.

Error-Code Detection For functions that return integers and make system calls, we employ fault injection at the system call level to expose return values that correspond to errors. We do so by re-running the application with $I_{\mathcal{F}}$, while intentionally failing all the system calls in the target function. If its return value changes in comparison to the previous run, we consider this new value an error code.

3) RP Selection: Finally, after collecting all functions satisfying **P1** and **P2**, we select the one closest to the *FS-*

TABLE III. APPLICATIONS AND FEATURES USED IN EVALUATION.

Application (Version)	**Feature Group** (Input format)	**Unwanted Feature** (CVE)
IMAGEMAGICK IM (7.0.9)	Image Support (file)	TIFF / SVG / PNG / JPEG / GIF (2019-15141, 2019-13136)
	Image Edit (UI)	Crop / Chop / Flop / Flip / Rotate / Shear
EVINCE (3.22.1)	File Operation (UI)	Print / Open / Save / Copy / Properties
BUSYBOX (1.22.0)	Applet Command (cmd line)	Wget / Bunzip2 / Gunzip / ... (102 more) (2018-1000500, 2018-1000517 2017-15873, 2015-9261)
EXIV2 (0.27.1.19)	Image Edit (cmd line)	Insert / Remove / Print / Extract / Rename
ZIP (3.0)	Zip Operation (cmd line)	-TT / -ds / -UN / -b / ... (5 more) (priviledge escalation [4])
NGINX (1.3.9)	Http Method (network data)	PUT / GET / MOVE / POST
	Transfer Encoding (network data)	Chunked Encoding (2013-2028)
PROFTPD (1.3.5e)	FTP Action (network data)	CPFR / CPTO / CHGRP / CHMOD (2015-3306)
EXIM (4.86)	Startup Script (cmd line/env. var.)	-ps & config file (2016-1531)
BASH (4.3)	Shell Function (env. var.)	Define functions with env. var. (2014-6271)

edge and its associated error code as RP. For example, for disabling NGINX's PUT method, F-BLOCKER defines function `ngx_epoll_process_events` and return value `-1`, as a rescue point. As shown in Fig. 5 this functions satisfies all three properties (**P1-P3**). While function `ngx_http_parse_request_line` would be an ideal RP, it does not perform any system calls, which does not allow us to infer a return value used to signal an error to its callers.

VI. IMPLEMENTATION AND EVALUATION

We have implemented our system on top of Intel's Pin [20] and IDA Pro [13] with around 4,000 lines of C++ code and 700 lines of Python code. Our system currently supports Linux platforms, but due to the interoperability of the tools we use, it can be extended for Windows platforms with minor effort. We evaluated F-DETECTOR across the following dimensions:

- **Correctness:** *Can* F-DETECTOR *detect the correct FS-edge for disabling features?*
- **Security:** *Can* F-DETECTOR *neutralize vulnerabilities by disabling functionality?*
- **Continuity:** *Can* F-BLOCKER *ensure continuity of the application by defining an appropriate rescue point?*

A. Setup and Benchmarks

To evaluate F-DETECTOR and F-BLOCKER, we ran experiments using 9 real-world applications. The selected applications span a wide spectrum of families (servers, utilities, shell programs, etc.). Among all the benchmark applications, we disable 146 of their features, which we selected randomly by studying their test suites and manuals. The features (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); (iii) are associated with various types of vulnerabilities. The chosen applications and features are listed in Table III. We also evaluate F-DETECTOR with Coreutils to allow for comparison with RAZOR.

B. Correctness: Finding the Right FS-edge

To disable the selected features, we obtain inputs that trigger them ($I_{\mathcal{F}}$) from test suites or define them based on examples in their manuals. We generate other inputs ($\neg I_{\mathcal{F}}$) through minimally mutating the first according to the guidance in §IV-A. If their are more than two inputs that activate the feature we repeat our test with *every* possible combination of input pairs. In each test, we prepare *several* mutations ($\neg I_{\mathcal{F}}$) of the used inputs based on our strategy. To test the effect of using a different number (M) of mutations, we test two setups $M = 2$ and $M = 3$. In each setup, when doing majority voting among the edges obtained by different mutations, we try out every combination of M-mutations to determine an *FS-edge*, gaining an understanding about the effects of using different mutations. For the 146 features, we manually verify the correctness of each uniquely identified *FS-edge* (i.e., whether they block $\mathcal{F}$ but no other feature). We summarize the inputs and their mutations in Table IV.

1) Results Overview: The results of our experiment are shown in Table V. For all tested features in IMAGEMAGICK-UI, EVINCE, ZIP, NGINX, EXIM, and BASH, F-DETECTOR found the correct *FS-edge* to fully disable the unwanted feature under all setups. For a few feature-app pairs (apps IMAGEMAGICK-file, EXIV2, and PROFTPD), we see that in a few cases majority cannot be reached with $M = 2$, but with $M = 3$. Eventually, majority is achieved with the exception of BUSYBOX, where F-DETECTOR may err. Even with BUSYBOX, for the majority of tested features (100), only a small portion of combinations (0.5% when M = 3) provided erroneous results. We will discuss the cause of errors later in this section. Overall, the results indicate that F-DETECTOR *produces correct results for the majority of applications and features.*

2) Case Studies: To understand what prevented F-DETECTOR from working correct in all cases, we performed an in-depth analysis of the cases where majority was not reached immediately or where there were errors. A similar analysis of the features, where we did not face problems, is presented in Appendix B.

- **IMAGEMAGICK:** We found that two of the GIF images in the test suite included animations. When we converted them to PNG (one of them) and TIFF (both of them) for use as $\neg I_{\mathcal{F}}$, they retained GIF-formatted data. Consequently, $\neg I_{\mathcal{F}}$ included inputs that still activated GIF-related functionality. Essentially, by not understanding what is in the test suite we employed, we did not correctly generate $\neg I_{\mathcal{F}}$.
- **EXIV2:** EXIV2 provides three different ways (essentially, aliases) to activate $\mathcal{F}$, as shown in Table IV, and it uses two separate parser routines to process them. If we only use two inputs that use the same parser, we cannot detect

TABLE IV. TEST CASES USED IN OUR EVALUATION.

Application	Feature	Inputs $I_{\mathcal{F}}$	Mutations $\neg I_{\mathcal{F}}$
IMAGEMAGICK (file)	TIFF	`Libtiff5` test cases (33)	`tiff` →{`jpg, png, gif`}
	PNG	`libpng` test cases (104)	`png` →{`tiff, jpeg, gif`}
	JPEG	`libjpeg` test cases (8)	`jpeg` →{`tiff, png, gif`}
	GIF	`giflib` test cases (3)	`gif` →{`tiff, png, jpeg`}
	SVG	`librsvg` testcase (21)	`svg` →{`tiff, png, jpeg, gif`}
IMAGEMAGICK (UI)	6 features	Click button for the feature	Click other buttons under same menu (5)
EVINCE	5 features	Click button for the feature; Typing keyboard shortcut for the feature	Click other buttons under same menu (5); Typing shortcut of other features (5)
BUSYBOX	105 features	e.g. `./busybox` **`wget`**	e.g. `wget` →{`bunzip2, ping, ... 104 others`}
EXIV2	5 features	e.g. `./exiv2` **`insert`** `image`; `./exiv2` **`in`** `image`; `./exiv2` **`-i a`** `image`	e.g. `insert` →{`delete, extract, ...`}; `in` →{`rm, ex, pr, mv`}; `-i a` →{`-d a, -e a, -p E, -r <fmt>`};
ZIP	9 features	e.g. `./zip file.zip file.txt -T -TT=ls`; `./zip file.zip file.txt -T --unzip-command=ls`	e.g. `-TT` →{`-b, ... 8 others`}; `--unzip-command` →{`--temp-path, ... 8 others`}
NGINX	4 features	e.g. `curl -X GET -D "" url`	e.g. `GET` →{`PUT, MOVE, POST`}
	Chk. Enc.	`curl -H "Transfer-Encoding: Chunked" url`	`Chunked` →{`Identify`}
PROFTPD	CPFR	{`SITE CPFR /tmp/file`}	`CPFR` →{`CPTO, CHGRP 777, CHMOD 777`}
	CPTO	{`SITE CPTO /tmp/file`}	`CPTO` →{`CPFR, CHGRP 777, CHMOD 777`}
	CHGRP	{`SITE CHGRP 777 /tmp/file`}	`CHGRP` →{`CPFR, CPTO, CHMOD 777`}
	CHMOD	{`SITE CHMOD 777 /tmp/file`}	`CHMOD` →{`CPFR, CPTO, CHGRP 777`}
EXIM	Startup sct.	**`-ps`** `/path/to/script` Enable in config file	`-ps` →{`-Ej, -bV, -dr, ... 6 others`} Disable in config file
BASH	Env. func.	`x="() { :; }"`	`x="{ :; }"` →{`"", "1"`}

TABLE V. EVALUATION OF F-DETECTOR. CORR IS THE % OF EXPERIMENTS WHERE THE CORRECT *FS-edge* IS DETECTED; NORES THE % WHERE MAJORITY CANNOT BE REACHED; AND INCORR THE % WHERE AN INCORRECT *FS-edge* IS DETECTED. FEATURES WITH IDENTICAL RESULTS ARE COLLAPSED INTO THE SAME ROW.

App	Feature	M = 2 Corr / Nores / Incorr	M = 3 Corr / Nores / Incorr
IM (file)	TIFF	100% / 0% / 0%	100% / 0% / 0%
	PNG	100% / 0% / 0%	100% / 0% / 0%
	JPEG	100% / 0% / 0%	100% / 0% / 0%
	GIF	65% / 45% / 0%	72% / 28% / 0%
	SVG	100% / 0% / 0%	100% / 0% / 0%
IM (UI)	All (6)	100% / 0% / 0%	100% / 0% / 0%
EVINCE	All (5)	100% / 0% / 0%	100% / 0% / 0%
BUSYBOX	100 feat.	91.5% / 9.3% / 0.2%	99.5% / 0% / 0.5%
	5 feat.	0.1% / 7.4% / 92.5%	0.3% / 0% / 99.7%
EXIV2	5 feat.	67% / 33% / 0%	67% / 33% / 0%
ZIP	All (9)	100% / 0% / 0%	100% / 0% / 0%
NGINX	4 feat.	100% / 0% / 0%	100% / 0% / 0%
	Ch.-Enc.	100% / 0% / 0%	—
PROFTPD	CPFR	33% / 67% / 0%	100% / 0% / 0%
	CPTO	100% / 0% / 0%	100% / 0% / 0%
	CHGRP	33% / 67% / 0%	100% / 0% / 0%
	CHMOD	33% / 67% / 0%	100% / 0% / 0%
EXIM	All (1)	100% / 0% / 0%	100% / 0% / 0%
BASH	All (1)	100% / 0% / 0%	—

the correct *FS-edge* where the paths converge. Instead an erroneous edge is detected. Listing 3 shows the edge when $\mathcal{F}$ is the `insert` command and `rename` is used as $\neg I_{\mathcal{F}}$. Again, we have erred by not ensuring that $I_{\mathcal{F}}$ includes *all* the aliases of the command-line option we are trying to disable. Nevertheless, majority voting protected F-DETECTOR from making the wrong decision.

```
1  int Params::nonoption(const std::string& argv){
2          ...
3          if (argv == "in" || argv == "insert") {
4            ...
5          }
6
7          (\emph{conditional jump}).
8          if (argv == "mv" || argv == "rename") {
9            ...
10         }
11         ...
12 }
```

Listing 3. Erroneous edge in EXIV2, when using `rename` in $\neg I_{\mathcal{F}}$.

- **PROFTPD:** The FTP features we are testing belong to two command categories [36]: *Direct File Duplication* (`SITE CPFR/SITE CPTO`) and *Owner or Group Change* (`CHGRP/CHMOD`). `CPFR` and `CPTO` are essentially sub-commands of the `SITE` command. PROFTPD parses all `SITE` commands by sequentially calling a function that tests each sub-command. Afterward another function tests for `CHGRP` and `CHMOD`. Because of this uncommon pattern, no *FS-edge* obtains majority when (i) `CPFR` and `CPTO` are used as $I_{\mathcal{F}}$ and $\neg I_{\mathcal{F}}$, respectively or (ii) when a command from the second group is targeted as $\mathcal{F}$ using just mutations from the first group. In this case, as well, the problem could be

avoided by carefully examining the available documentation for the targeted features, as all variations would succeed, if $I_{\mathcal{F}}$ included commands both from the same and other groups. Majority voting perseveres over this mistake.

- **BusyBox:** F-DETECTOR finds the correct *FS-edge* for most of the features (100/105) in most of the tested combinations. The *FS-edge* is routed on the same instruction because it corresponds to an `icall` to the function implementing the unwanted applet. The errors encountered occurs when five applets (`ping`, `traceroute`, `ping6`, `traceroute6`, and `crontab`) are used as inputs with other applets used as mutations and vice versa. These five applets drop SUID privileges, while others do not, which leads to erroneously detecting an edge that deals with SUID as the *FS-edge*, as shown in Listing 4. This edge disables all features based on whether they require or drop SUID privileges. If the mutation function generated $\neg I_{\mathcal{F}}$ with the same SUID requirements as $I_{\mathcal{F}}$, these errors could be avoided. Essentially, the minimal mutation strategy is inadvertently not working.

```
1 /* FS-edge is the true branch of the conditional statement
      checking whether the applet needs privilege or not*/
2 if ( APPLET_SUID(applet_no) == BB_SUID_REQUIRE ) {
3    ... /* privileges needed in 104 applets*/
4 } else if (APPLET_SUID(applet_no) == BB_SUID_DROP) {
5   ... /* drop all privileges in 5 applets */
6 }
```

Listing 4. Wrong *FS-edge* in BusyBox (SUID-dropping applet in $\neg I_{\mathcal{F}}$).

3) Effects of Mutations and Inputs: Our evaluation reveals that producing correct results can be influenced by the inputs ($I_{\mathcal{F}}$) and mutations ($\neg I_{\mathcal{F}}$) used to disable $\mathcal{F}$. Note that prior works are also affected by the type of test cases used, however, this aspect is mostly ignored. Overall, we have learned the following from evaluating F-DETECTOR:

- Increasing the number of mutations increases robustness.
- Simple inputs are better (e.g., non-animated GIFs).
- Using inputs ($I_{\mathcal{F}}$) that include all aliases of a command-line option, protocol field, etc. increases robustness.

These rules-of-thumb augment the minimal mutation guidelines, we defined earlier.

4) When Mutations and Majority Voting Fail: Our tests of BusyBox were extensive, disabling a large number of applets and mutating each $I_{\mathcal{F}}$ to all other available applets. Our experiment revealed that, even though using multiple mutations is beneficial, we should *not* sacrifice the minimal mutation requirement for quantity. BusyBox provides numerous options for mutating a feature-activating test case. If we only mutated to similar applets, e.g., file-utilities to file-utilities, network-utilities to network-utilities and so on, errors could be significantly reduced, if not eradicated.

C. Evaluating F-DETECTOR on Coreutils

As discussed in §II, the methodology used by approaches like RAZOR can lead to over- or under-debloating. We test F-DETECTOR on the same benchmark programs as RAZOR data to determine, if we can remove features without similar issues. We consider the features used for train in RAZOR as the wanted features. We select some from the remaining ones (not trained) as unwanted features to disable (listed in Table VI in the appendix). In summary, we disabled 2 feat. in `bzip2`, 5 in `chown`, 10 in `date`, 8 in `grep`, 2 in `gzip`, 1 in `mkdir`, 3 in `rm`, 10 in `sort`, 7 in `tar`, and 1 in `uniq`. After disabling each of the unwanted features, we verify that the functionality is fully disabled and none of the wanted features is affected. The results show that F-DETECTOR disabled all features without any over- or under-debloating problems.

D. Security Benefits of Feature Removal

To understand the security benefits of F-DETECTOR, we tested whether we can effectively mitigate vulnerabilities associated with unwanted features by disabling them. First, we prepared a proof-of-concept (PoC) input to trigger each of the 10 vulnerabilities listed in Table III. We ran the PoC against the unmodified applications and on versions where the *FS-edge* stops execution. In all cases, the *FS-edge* caused an application exit, before the PoC reached vulnerable code.

To verify that the *FS-edge* completely mitigates the CVE and not just the PoC, we manually analyzed each vulnerability. We found that all the vulnerabilities lie in code only reachable via the *FS-edge*. Below we present a detailed analysis of CVE-2013-2028 (associated with the "Chunked Encoding" feature of NGINX) and how it is mitigated by F-DETECTOR. Other vulnerabilities are disabled in similar manner.

The vulnerability is a stack buffer overflow in function `ngx_http_parse_chunked` [30]. This function is called by four functions:

① `ngx_http_discard_request_body_filter`
② `ngx_http_request_body_chunked_filter`
③ `ngx_http_proxy_chunked_filter`
④ `ngx_http_proxy_non_buffered_chunked_filter`

Listing 5 shows that `ngx_http_parse_chunked` is *only* called if `r->headers_in.chunked` is true, in all cases. The only location setting `r->headers_in.chunked` to true is the destination basic block of the *FS-edge* detected by F-DETECTOR (shown in Listing 10). As our design guarantees that the destination of the *FS-edge* is uniquely reachable through it, setting `r->headers_in.chunked` to true is disabled and the vulnerability is neutralized.

```
1  int ngx_http_discard_request_body_filter(...){①
2     if (r->headers_in.chunked) {
3        rc = ngx_http_parse_chunked(r, b, rb->chunked);
4        ...
5     }
6  }
7  int ngx_http_request_body_filter(...){
8     if (r->headers_in.chunked) {
9        return ngx_http_request_body_chunked_filter(r, in);②
10    }
11    ...
12 }
13 int ngx_http_proxy_input_filter_init(...){
14    if (u->headers_in.chunked) {
15       u->pipe->input_filter =
16           ngx_http_proxy_chunked_filter;③
17       u->input_filter =
               ngx_http_proxy_non_buffered_chunked_filter;④
18       ...
19    }
20 }
```

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

E. 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 [25]. 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 VII in the appendix. Below, we present our analysis of the NGINX and PROFTPD RPs.

1) NGINX*:* When a PUT request is received, the *FS-edge* in `ngx_http_process_request_line` causes a fault which triggers a rollback to the RP function, `ngx_epoll_process_events`, which is higher in the call stack. The RP returns error code (`-1`), which was identified through system call-based fault injection. As shown by the function's source code (line 4 in Listing 6), this is indeed a valid error code returned by the function. Interestingly, the caller of the RP, `ngx_process_events_and_timers`, ignores its return value intentionally (line 11). However, by design, ignoring the poll event corresponding to the PUT request is sufficient to resume processing other sockets on the next loop iteration, while the offending socket is eventually closed and discarded.

```
1  static ngx_int_t ngx_epoll_process_events(...) {
2      ...
3      events = epoll_wait(...);
4      err = (events == -1) ? ngx_errno : 0;
5      ...
6  }
7  void ngx_process_events_and_timers(...) {
8      ...
9      /* ngx_process_events --> ngx_epoll_process_events */
10     (void)ngx_process_events(cycle, timer, flags);
11     ...
12 }
```

Listing 6. NGINX RP function for PUT method feature.

2) PROFTPD*:* When a `CPFR` command is received, the *FS-edge* in `copy_cpfrm` triggers a rollback to the beginning of the RP function, which is also `copy_cpfrm`. F-BLOCKER determined that the RP returns pointers and assigned a NULL error value to it. As shown in Listing 7, the RP indeed returns a pointer (line 11), which is checked by its caller, `_dispatch`, propagated to its own caller, and, eventually, results in returning an error message to the user.

```
1  static int _dispatch(...) {
2      ...
3      mr = pr_module_call(c->m, c->handler, cmd);
4      ...
5      if (!mr && !success && validate) {
6          ...
7          success = -1;
8      }
9      return success;
10 }
11 modret_t *pr_module_call(...) {
12     /* func is a pointer pointing to copy_cpfrm */
13     res = func(cmd);
14     ...
15     return res;
16 }
```

Listing 7. PROFTPD RP function for `CPFR` command.

VII. RELATED WORK

A. Debloating at the Source-code Level

Perses [35] and C-Reduce [28] are the state-of-the-art program reduction tools that build upon the concept of delta debugging [37], [21]. By specifying a program to be minimized and an arbitrary property test function, these tools return a minimized version of the input program that is also correct with respect to the given property. Chisel [12] further improves this approach, by leveraging reinforcement learning. Via repeated trial and error, Chisel builds a model to determine the likelihood of a candidate, minimal program to pass the property test.

Chisel uses input-based specification similar to ours. However, it requires that inputs to activate all desired functionality are defined. Generating high-coverage inputs to all desired functionality is hard, as attested to by the low-coverage by developer-written tests and the continuing efforts in improving the coverage of fuzzing systems. In contrast, our approach just needs a small number of inputs and a small set of guided mutations. In addition, CHISEL utilizes statistical methods, which may over fit the application to the test inputs and aggressively remove actually needed functionality. Instead, F-DETECTOR is geared towards disabling specific functionality and thus, disables smaller targeted parts of the application. Finally, CHISEL operates on source code. Unlike our approach, CHISEL is not appropriate for binary programs where semantic information was stripped by compilation.

B. Debloating Java Bytecode

JRED [15] debloats Java applications and the Java Runtime Environment. It operates on Java bytecode and performs conservative static analysis to understand reachability, followed by binary rewriting to remove unreachable methods and classes. JRED removes unused (instead of unwanted) functionality. Jiang et al. [16] consider "the methods of interests" as seed methods and define a feature as all call sites of the seed methods. To cut a feature, they remove call sites of the seed methods as well as the resulted redundancy. Correctness of this approach heavily depends on the accuracy of the seed

methods, bringing significant burdens to users. Test-based Software Minimization [8], [7] (TBSM) removes unwanted functionality based on developer-defined, annotated test cases. The authors, while working with developers, observed that they "speak the language of tests fluently", unlike that of formal methods or architectural descriptions. Tests are, thus, potentially more practical in defining unwanted functionality. 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.

C. Debloating at the Binary-code Level

BlankIt [24] focus on activating library function on demand, instead of removing unwanted functionality. BlankIt runs static analysis to predict the call targets, which can result in mispredictions hurting program functionality. BlankIt's on-demand loading of code leaves a window for attackers to load needed code by hijacking control-flow and reusing trampolines. BlankIt also imposes a significant run-time overhead, even with programs that do not heavily use libraries. Instead of directly working on the code, Quach et al. [27] approach software debloating by relying on the compiler and the linker. They first tailor the compiler to analyze dependencies across components that are necessary to ensure execution. Then they customize the loader to eliminate code disconnected to the dependency graph. Similar to [27], Nibbler [2] de-bloats software by removing unused functions from the linked libraries. The difference is Nibbler only uses binary information.

Chen et al. [5] take definitions and approaches similar to [16]. They require information of a seed function to uniquely represent a feature and define constituent functions on the paths leading to the seed function as the feature. They further insert gates into the constituent functions pertaining to desired features, which prevent the execution from escaping the gated functions. A follow-up work [6] improves the identification of constituent functions with symbolic execution, particularly targeting network-based applications.

Ghaffarinia and Hamlen [9] assume that the users can demonstrate desired features via unit tests. Using the sub-CFG constructed in unit tests as a basis, they leverage machine learning to derive a more complete, probabilistic contextual CFG (a CFG that carries all permitted control flow transfers). Finally, they statically instrument the binary to prevent deviation from the contextual CFG. Razor [26] is another approach to retain wanted features. Since we have discussed it in § II, we omit the details.

Despite all aim to eliminate unwanted features, F-DETECTOR has several differences from the above approaches. First, F-DETECTOR directly removes the unwanted features while those approaches retain the desired functionality. It is more straightforward and feasible to gradually identify the list of unwanted features than determining the desired features at once. Second, those approaches require domain knowledge to prepare a list of "seed functions" or a comprehensive corpus of test suites. In contrast, F-DETECTOR only requires a few inputs and several guided mutations from the user, posing less burdens to the user.

More closely related to our work, Landsborough et al. [19] develop several early-stage approaches to removing features, considering the instruction traces following a specific group of inputs as a feature. Technically, they collect instruction traces with test inputs for both desired features and unwanted features. They rewrite the code that is never reached with *nops*. They also overwrite the code activated by unwanted features but not desired features. These approaches largely rely on the comprehensiveness of test inputs, which can easily result in accidental removal of functional code.

D. Vulnerability Workaround

To mitigate vulnerabilities prior to patching, an orthogonal approach of feature removing is vulnerability workaround. Huang et al. propose TALOS [14] to create security workarounds as a rapid response to disclosed vulnerabilities. TALOS works by redirecting all execution paths that reach the vulnerable code to built-in error handling code. Unlike our approach that relies on dynamic information, TALOS relies on static analysis to identify error-handling code within the applications. TALOS was later extended by RVM [38] to handle binary-only programs. In comparison to TALOS and RVM, our approach has the advantage of preventing functionality loss. This is because TALOS and RVM block all execution reaching the vulnerable code, regardless of the execution contexts (where functionality is being used). In contrast, our approach only disables the specific feature, without affecting other functionality.

VIII. Conclusion

This paper presents a novel solution for disabling unwanted features, as a means of reducing attack surface. Unlike previous research that depends on burdensome user specifications to determine *all* desired features, our solution only requires a small set of inputs to activate the *unwanted* feature and guided, minimal mutations on the inputs to avoid the unwanted feature. We combine dynamic tracing on the provided inputs and static analyses to identify a single control-flow edge that dominates the feature as its entrance. Going beyond, we leverage error virtualization to disable the unwanted feature safely by redirecting execution on the feature entrance to built-in error handling. This ensures the continuity of normal service in the target program. We have implemented our solution on the Linux platform and evaluated it with 146 features from 9 programs. Our solution detected and disabled all features, under all settings, except for a few BUSYBOX features. This demonstrates the utility of our solution in improving security for a wide-range of binary applications and sheds light on future research.

References

[1] "The Heartbleed bug," https://heartbleed.com/.

[2] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. USA: ACM, December 2019, pp. 70–83.

[3] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman, "Design and implementation for checkpointing of distributed resources using process-level virtualization," in *International Conference on Cluster Computing (CLUSTER)*. USA: IEEE, 2016, pp. 402–412.

[4] R. Chandel, "Linux for Pentester : ZIP Privilege Escalation," https://www.hackingarticles.in/linux-for-pentester-zip-privilege-escalation/, 2019.

[5] Y. Chen, T. Lan, and G. Venkataramani, "DamGate: Dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*. ACM, 2017, pp. 23–29.

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

[7] A. Christi, A. Groce, and R. Gopinath, "Resource adaptation via test-based software minimization," in *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2017, pp. 61–70.

[8] A. Christi, A. Groce, and A. Wellman, "Building resource adaptations via test-based software minimization: Application, challenges, and opportunities," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 73–78.

[9] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 1009—-1022.

[10] W. Glozer, "WRK: Modern HTTP benchmarking tool," https://github.com/wg/wrk, 2019.

[11] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *Journal of Physics: Conference Series*, pp. 494–499, sep 2006.

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

[13] Hex-Rays, "The IDA pro disassembler and debugger," https://www.hex-rays.com/products/ida/, 2020.

[14] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *Symposium on Security and Privacy (SP)*. IEEE, May 2016, pp. 618–635.

[15] Y. Jiang, D. Wu, and P. Liu, "JRed: Program customization and bloatware mitigation based on static analysis," in *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*. USA: IEEE, June 2016, pp. 12–21.

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

[17] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, September 2002.

[18] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the European Workshop on Systems Security (EUROSEC)*. ACM, 2019, pp. 9:1–9:6.

[19] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *Companion Publication of the Annual Conference on Genetic and Evolutionary Computation*. USA: ACM, 2015, pp. 833–838.

[20] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 190–200.

[21] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006, pp. 142–151.

[22] N. P. Myhrvold, "The next fifty years of software," http://hartenstein.de/EIS2/next50years.pdf, 1997.

[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 361–376, Dec 2002.

[24] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you don't," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 164–180.

[25] G. Portokalidis and A. D. Keromytis, "REASSURE: A self-contained mechanism for healing software using rescue points," in *Proceedings of the International Workshop on Security (IWSEC)*. Berlin, Heidelberg: Springer-Verlag, November 2011, pp. 16–32.

[26] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *USENIX Security Symposium*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1733–1750.

[27] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *Proceedings of the USENIX Security Symposium*. USA: USENIX Association, Aug 2018, pp. 869–886.

[28] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case Reduction for C Compiler Bugs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 335–346.

[29] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *The International Journal of High Performance Computing Applications*, pp. 479–493, 2005.

[30] SecurityFocus, "Nginx 'ngx_http_parse.c' Stack Buffer Overflow Vulnerability," https://www.securityfocus.com/bid/59699, 2013.

[31] Selectel, "ftpbench," https://github.com/selectel/ftpbench, 2014.

[32] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: Application specialization for code debloating," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 329–339.

[33] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "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, 2009, p. 37–48.

[34] Standish Group, "CHAOS report 2009," https://www.classes.cs.uchicago.edu/archive/2014/fall/51210-1/required.reading/Standish.Group.Chaos.2009.pdf, 2009.

[35] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided Program Reduction," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 361–371.

[36] WinSCP, "Supported file transfer protocols," https://winscp.net/eng/docs/protocols, 2020.

[37] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, pp. 183–200, Feb. 2002.

[38] H. Zhen and T. Gang, "Rapid vulnerability mitigation with security workarounds," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*. Reston, VA, USA: ISOC, 2019.

[39] T. Zimmermann, N. Nagappan, and L. Williams, "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, 2010, pp. 421–428.

APPENDIX 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. 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
```

B. 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
```

APPENDIX B
ANALYSIS OF REMAINING *FS-edges*

IMAGEMAGICK-file: In the test of features supporting TIFF / SVG / PNG / JPEG, we detected the same, proper *FS-edge* with any combination of 2-inputs and M-mutations. The *FS-edge*, as shown in Listing 1 (in the paper), 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-inputs and M-mutations. Listing 8 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, providing both completeness and safety.

```
/* 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 8. *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-inputs and M-mutations. The *FS-edge* is an indirect call to the function implementing the feature. The related code is in Listing 9.

```
/*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 9. *FS-edge* in `Evince` for Print feature.

NGINX: By using any 2-inputs 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 10, to disable the "Chunked Encoding " feature without hurting other functionality (only one combination of 2-inputs and 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.

```
/* 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 &&
```

```
5       ngx_strncasecmp(r->headers_in.transfer_encoding
6       ->value.data,(u_char *) "chunked", 7) == 0 ){
7          r->headers_in.content_length = NULL;
8          r->headers_in.content_length_n = -1;
9          r->headers_in.chunked = 1;
10         }
11      ...
12 }
```

Listing 10. *FS-edge* in NGINX for the "Chunk-Encoding" feature.

ZIP: The tests with ZIP produced similar results to IMAGEMAGICK-UI. We detected the correct *FS-edge* for every feature, regardless, no matter which 2-inputs and M-mutations we used. The detected *FS-edge* for all the features follows the same pattern in Listing 8: an indirect jump from the switch checking the command line to the case implementing the target feature.

EXIM: We detected the *FS-edge* shown in Listing 11 for the "startup script" feature in EXIM, using any combination of 2-inputs and 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.

```
1 /* FS-edge is a conditional jump to the code launching the
      startup script. */
2 int main(int argc, char **cargv){
3  ...
4    if (opt_perl_at_start && opt_perl_startup != NULL ){
5   errstr = init_perl(opt_perl_startup);
6    .../*code handling Chunked Encoding''*/
7  }
```

Listing 11. *FS-edge* in EXIM for the "startup script" feature.

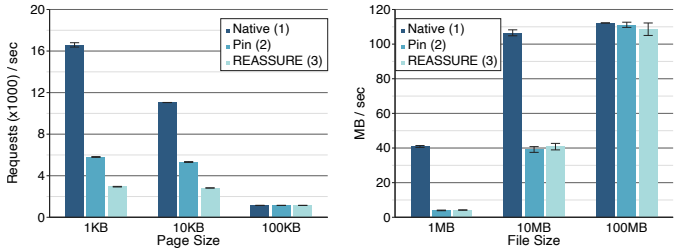
BASH: We detected the *FS-edge* shown in Listing 12 for the "defining functions by environment variable" feature in BASH, with any 2-inputs and M-mutations. The *FS-edge* is a conditional jump to the code reading and executing the functions defined in the environment variable. Cutting off the *FS-edge* will prevents execution of those function without harm to any other functionality.

```
1 /* FS-edge is a jump to the code reading and executing the
      function defined in the environment variable"*/
2 void initialize_shell_variables (...){
3    ...
4  if (privmode == 0 && read_but_dont_execute == 0 &&
       STREQN ("() {", string, 4) ){
5       string_length = strlen (string) ;
6       ...
7    }
8 }
```

Listing 12. *FS-edge* in BASH for "functions in env. variables" feature.

APPENDIX 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 [11], [29], [23], [3], [33].



(a) NGINX (b) PROFTPD

Fig. 6. Performance of NGINX and PROFTPD with and without REASSURE. (a) We used WRK [10] to measure requests / second with different page sizes. (b) We used `ftpbench` [31] 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.

APPENDIX D
MISCELLANEOUS

TABLE VI. EVALUATING F-DETECTOR ON RAZOR'S BENCHMARK DATA (COREUTILS). THE TABLE LISTS THE FEATURES TRAINED IN RAZOR'S EXPERIMENTS AS THE WANTED FEATURES AND THE REMAINING ONES (NOT TRAINED) AS UNWANTED FEATURES.

Application	Wanted Features	Unwanted Features
bzip2-1.0.5	--compress	--decompress --test
chown-8.2	--recursive, --no-dereference	--changes --verbose --from --reference --no-preserve-root
date-8.21	+%c, +%d, +%D, +%F, and 9 others	+%A +%a +%b +%B 6 others
grep-2.19	--regexp, --extended-regexp	--basic-regexp --perl-regexp --word-regexp --line-regexp 4 others
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, --zero-terminated, --stable	--ignore-case --month-sort --numeric-sort --random-source 6 others
tar-1.14	--create	--list --extract --compare --append 3 others
uniq-8.16	--count, --repeated, --skip-fields, and 4 others	--zero-terminated

TABLE VII. RESCUE POINTS GENERATED BY F-BLOCKER. FOR RP DISTANCE PAIRS (A)/(B): THE VALUES CORRESPOND TO (A) THE DEPTH OF THE FUNCTION CONTAINING THE *FS-edge* AND (B) THE DEPTH OF THE RESCUE POINT FUNCTION IN THE CALL TRACE (DISTANCE FROM `__LIBC_START_MAIN()`).

Feature	RP Distance	Detection Method	Error Returned	RP Function Name
IMAGEMAGICK (file)	6/8	Pointer return	0 (NULL)	GetMagickInfo
IMAGEMAGICK (UI)	3/5	Syscall failing	0 (old value = 1)	DisplayImageCommand
EVINCE	23/28	Pointer return	0 (NULL)	g_action_group_activate_action
EXIV2	4/4	Pointer return	0 (NULL)	Action::Insert::clone_()
NGINX	7/10	Syscall failing	-1 (old value = 0)	ngx_epoll_process_events
PROFTPD	8/8	Pointer return	0 (NULL)	copy_cpfr
BUSYBOX	6/6	Syscall failing	1 (old value = 0)	wget_main
EXIM	1/1	Syscall failing	1 (old value = 0)	main
BASH	1/3	Syscall failing	1 (old value = 0)	main
ZIP	1/1	Syscall failing	1 (old value = 0)	main