Disabling Unwanted Functionality in Binary Programs

Anonymous Author(s)

1 Disabling Features Safely with Rescue Points

Given an FS-edge, we can easily disable $\mathcal{F}$ by terminating the application when the edge is traversed (e.g., by injecting a single-byte instruction like `int3`). However, this is undesirable for servers and long-running applications, because it reduces availability and may cause data loss. To disable $\mathcal{F}$ without terminating, we treat FS-edges as unanticipated errors and leverage software self-healing techniques based on rescue points and error virtualization [13, 14] to handle them.

Rescue points (RPs) are functions that enable the virtualization of unanticipated errors by returning another valid error code, which the application can handle to gracefully recover and continue. The workflow is presented in Fig. 1. At a high level, the RP function contains an execution path to FS-edge. Upon entry, a checkpoint (or snapshot) of the process or system state is created. Traversing the FS-edge triggers a fault, which causes a rollback to the checkpoint state. Finally, the RP function returns a valid error code to the application. If the FS-edge is not hit, the checkpoint state is released upon return of the RP function.

In this section, we discuss F-BLOCKER which automatically identifies and configures RPs for FS-edges. The system design is depicted in Fig. 2. Our goal is to generate RPs for existing self-healing systems [10, 13] that meet the following criteria:

- **P1**: Every path that can lead to the FS-edge goes through the RP, so we can always "rescue" the application. For instance, the function containing the FS-edge always satisfies this criterion.
- **P2**: The RP returns an error code that triggers built-in error handling. This is a key piece of error virtualization.
- **P3**: The RP should be close to the FS-edge. This narrow down the scope of the checkpoint and reduces overhead.

1.1 Call-Trace Extraction

We start by running the application with the feature-triggering inputs that were used with F-DETECTOR ($I_{\mathcal{F}_u}$). 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.

1.2 Rescue-Point Generation

Dominance Analysis. To satisfy **P1**, the RP must be one of the functions in the call stack obtained in the first step. To 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

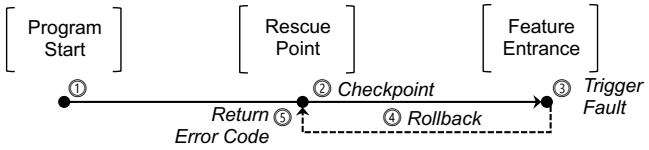

Figure 1: Workflow of software self-healing with rescue points.

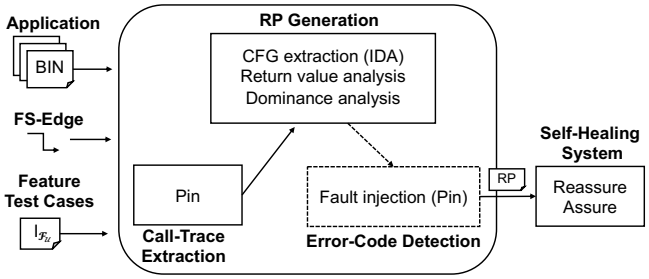

Figure 2: F-BLOCKER overview.

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.

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` are 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 [6, 15]. 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 use 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 $I_{\mathcal{F}_u}$, 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.

RP Selection. Finally, after collecting all functions satisfying **P1** and **P2**, we select the one closest to the FS-edge (the closest possible being the one containing it) and its associated error code as RP. In our Nginx example, the FS-edge is in `ngx_http_parse_request_line()`. Looking at the call stack below, the first function matching P1-P3 is `ngx_epoll_process_events()`, becoming the RP with error code -1.

```
ngx_http_parse_request_line    error code=<unknown>
ngx_http_process_request_line  <void function>
ngx_http_init_request          <void function>
ngx_epoll_process_events       error code =-1
ngx_process_events_and_timers  <void function>
ngx_worker_process_cycle       <void function>
        ...
```

1.3 Implementation

We have implemented the F-BLOCKER on top of Intel's Pin [8] and IDA Pro [5]. 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.

Altering Behaviors of System Calls. To select the rescue point, our analysis needs to intentionally fail system calls. In our implementation, we register callbacks at the entry and exit of each system call with Pin's API. When the system call is entered, we change the system call number to 0x101 (corresponding to *openat*) and we also modify the argument of the file name as an empty string. This ensures the system call to fail and return `-1` to signal an error.

Error Virtualization. We implemented the error virtualization by customizing REASSURE [10]. Once the rescue point is hit, we save the execution contexts (mainly the registers) and record every memory write by its address and the original value. When the FS-edge is reached, we rollback all recorded memory writes via rewriting their original values back. Then we restore the execution context at the rescue point, set up `eax/rax` to the predetermined error value, and configure the program counter to the return address.

1.4 Evaluation of F-BLOCKER

Following the detection of the FS-edge, we applied F-BLOCKER to disable the unwanted features. We observed whether the features can still be activated and whether the target programs can be healed. As summarized in Table 1, all features could no longer be activated. In all programs except EVINCE and IMAGEMAGICK (crop feature), the target programs were healed and remained in a stable state after error virtualization. Table 2 lists all RPs generated by F-BLOCKER.

1.4.1 Classification of Resulted Behaviors Table 1 presents the behaviors in the target programs after we disabled the unwanted features.

Table 1: Results of disabling unwanted features.

Feature	Healed	Resulted behavior
IMAGEMAGICK	✓	Console: Error message and application exit. UI: Error message popup.
IMAGEMAGICK (Crop)	✗	Crash (*Segmentation Fault*).
EVINCE	✗	Crash (*X11 server state cannot be rolled back.*)
EXIV2	✓	Returned error causes a SIGABRT after a failed assertion.
NGINX	✓	Server drops the request and client receives no message.
PROFTPD	✓	Client receives error message: *command not understood*.
BUSYBOX	✓	Application exits with error return code.
EXIM	✓	
BASH	✓	
ZIP	✓	

- In 6 cases, the program exited with an error code returned. In most of those cases, the FS-edge is inside the `main` function.
- IMAGEMAGICK returned an error indicating that the format is not supported, displayed as a pop-up window in the UI or message in the terminal.
- PROFTPD returned an error code `500` with a message showing that the command cannot be interpreted.
- NGINX was able to handle the error by updating the global state. However, the user did not receive any error message because NGINX expects a finalized request before sending an error message.
- EXIV2 raised a SIGABRT at a failed assertion. The reason behind is that the error virtualization returns a NULL pointer to the class representing the requested feature.

In all the above cases, our error virtualization resulted in either expected error handling or a reasonable alternative. This strongly supports that F-BLOCKER can reliably and safely disable unwanted features.

.

1.4.2 Understanding of Errors In two cases, F-BLOCKER led to crashes of the target programs. The failures were caused by the rollback process. In the case of EVINCE, F-BLOCKER was unable to recover the state of the *X11* server as it cannot rollback system calls. The error with IMAGEMAGICK was due to the freeing of an invalid pointer, which was also caused by unrecoverable system calls. To fix such errors, a quick solution is to adopt OS-level checkpoint-rollback techniques like those presented in [1, 2, 4, 7, 9, 11].

1.5 Impact to Performance

To safely disable a feature, F-BLOCKER uses checkpoint-and-rollback. Intuition suggests that this will incur performance overhead, nevertheless, we endeavor to minimize the range of tracing by defining a

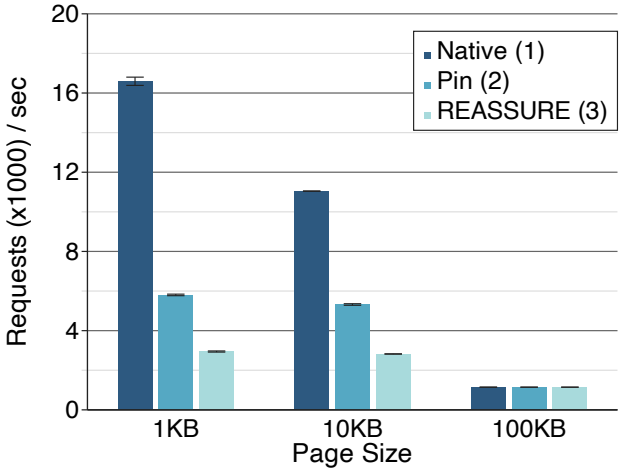

Figure 3: Nginx performance with and without F-BLOCKER for the *chunked encoding* feature. We use WRK [3] to measure requests/second with different file sizes over an 1Gb/s link.

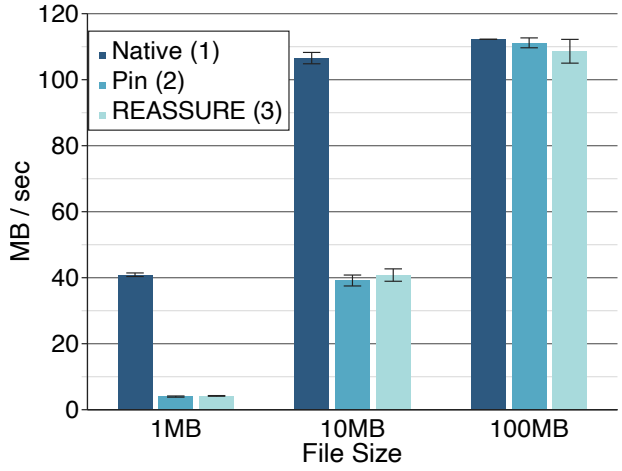

Figure 4: ProFTPD benchmark with and without F-BLOCKER. We use ftpbench [12] to measure the throughput with different files sizes over an 1Gb/s link.

RP near the FS-edge. Here, we present the results of our performance overhead evaluation.

We conducted the evaluation with NGINX and PROFTPD, considering they are widely used as performance benchmarks. To be specific, we ran NGINX and PROFTPD on a 2-core Xeon E3-1270 V2 @ 3.50GHz Xen VM with 29GB of RAM (Linux Debian 4.9.168-1+deb9u5, Xen 4.1). The benchmark clients were run on another host, a 4-core Xeon E3-1270 v6 @ 3.80GHz and 64GB of RAM (Ubuntu 16.04.6 LTS, Linux 4.8.0), 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 F-BLOCKER deployed. The experiments were repeated five times, and we draw the mean and standard deviation (SD) in Figures 3 and 4.

In the evaluation based on requests of small files (1KB), F-BLOCKER incurs x5.6 and x1.98 overhead, respectively considering the native execution and Pin as the baseline. When the file size increases to 100KB, we observe no significant overhead. This is potentially because the bottleneck moves from the CPU to the network, as well as because the frequency of requests is lower (but they take longer). The overhead is largely because the rescue point for the *chunked encoding* feature sits on a critical path, which is activated in nearly every request. Such scenarios represent the maximal level of impact that F-BLOCKER can bring to the normal execution.

In the case of PROFTPD, F-BLOCKER adds 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 F-BLOCKER heavily depends on the unwanted features and the correlation between the unwanted features and other features. For many of the cases, it can only negligibly impact performance. Going beyond, a large portion of the overhead is tied to the use of Pin, which can be largely mitigated by moving to OS-level checkpoint/rollback frameworks (e.g., [2, 4, 9, 11]).

References

[1] J. Adam, M. Kermarquer, J. B. Besnard, L. Bautista-Gomez, M. Perache, P. Carribault, J. Jaeger, A. D. Malony, and S. Shende. 2019. Checkpoint/restart approaches for a thread-based MPI runtime. *Parallel Comput.* 85 (2019), 204 – 219.

[2] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman. 2016. Design and Implementation for Checkpointing of Distributed Resources Using Process-Level Virtualization. In *International Conference on Cluster Computing (CLUSTER)*. IEEE, USA, 402–412.

[3] Will Glozer. 2019. WRK: Modern HTTP benchmarking tool. https://github.com/wg/wrk.

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

[5] Hex-Rays. 2020. The IDA Pro Disassembler and Debugger. https://www.hex-rays.com/products/ida/.

[6] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Symposium on Security and Privacy (SP)*. IEEE, 618–635.

[7] O. Laadan and J. Nieh. 2007. Transparent Checkpoint-restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 25:1–25:14.

[8] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200.

[9] S. Osman, D. Subhraveti, G. Su, and J. Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. *SIGOPS Oper. Syst. Rev.* 36 (Dec 2002), 361–376.

[10] Georgios Portokalidis and Angelos D. Keromytis. 2011. REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points. In *Proceedings of the International Workshop on Security (IWSEC)* (Tokyo, Japan). Springer-Verlag, Berlin, Heidelberg, 16–32.

[11] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. 2005. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *The International Journal of High Performance Computing Applications* (2005), 479–493.

[12] Selectel. 2014. ftpbench. https://github.com/selectel/ftpbench.

[13] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. 2009. ASSURE: Automatic Software Self-Healing Using Rescue Points. In

Table 2: 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`

Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 37–48.

[14] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. 2005. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, USA, 149–161.

[15] H. Zhen and T. Gang. 2019. Rapid Vulnerability Mitigation with Security Workarounds. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*. ISOC, Reston, VA, USA.