Uffd-monitor: A Userspace Code Execution Monitor for Minimal Executable Code

HIMANSHU DONGRE**, University of Illinois Chicago, USA

Large software systems suffer from ever-expanding codebases that increase the attack surface and provide more opportunities for vulnerabilities. Traditional approaches to attack surface reduction, such as static code debloating or address space layout randomization, are limited: static debloating is inflexible (removed code might be needed later, or new usage scenarios may break), while randomization retains all code in memory, merely shuffling it. We propose uffd-monitor, a userspace tool that implements code debloating by dynamically modifying the software runtime. Uffd-monitor enforces a strict budget of k code pages in memory at any time. Leveraging Linux's `userfaultfd` mechanism for userspace page fault handling and the `madvise` system call for evicting pages, uffd-monitor maintains a sliding window of code visibility in a running program. Pages of executable code are loaded on demand and evicted when not in use, so that only the currently needed code pages are mapped in memory. This design preserves full functionality while drastically reducing the instantaneous code available to an attacker, thereby reducing available gadgets and vulnerabilities. We implemented uffd-monitor on Linux 5.15 (x86-64) and evaluated its performance. The results show that dynamic code debloating is practical: our prototype imposes modest overhead while significantly shrinking the in-memory code footprint. In summary, this work introduces a flexible, adaptive tool that combines operating system paging capabilities with userspace monitoring for code debloating and security-hardened software runtime.

1 Introduction

Modern software contains far more code than any single execution will ever use, and this code bloat directly contributes to security risk. Large applications and libraries are packed with features, of which only a small fraction might be exercised in a given run. Studies have shown, for example, that on average only about 5% of the standard C library's code is actually used by programs—the rest is essentially latent functionality that bloats the binary. This extraneous code may harbor latent bugs or vulnerabilities and thus broadens the overall attack surface of the software. Meanwhile, the number of software vulnerabilities discovered each year continues to climb, with over 40,000 CVEs reported in 2024 alone. Reducing the amount of code exposed to attackers at runtime is a critical step toward mitigating this deluge of security issues.

One approach to tackle code bloat is *software debloating*, which removes or omits code that is not needed. Static debloating techniques, whether through ahead-of-time specialization, compiler-level dead code elimination, or post-hoc binary trimming, can indeed strip out a large portion of unused code. Recent work demonstrated that eliminating 70–80% of code from utility programs is often possible without breaking functionality. However, static debloating is fundamentally limited by its one-time, coarse-grained nature. Once a binary is debloated and deployed, its code footprint is fixed. If the debloating process was too aggressive, functionality needed later (or under different inputs) might be missing, causing failures. On the other hand, a conservative static debloating (to avoid breakage) may leave in place many chunks of code that won't be used in most runs, thereby failing to maximize the security benefit. A key shortcoming of static methods is that they ignore the time dimension of program execution—they cannot adapt to changing usage patterns or dynamically enable/disable features. Additionally, static debloating often cannot remove unused code in shared libraries, since these are pre-compiled and used by many programs; the dynamic linker will load the entire library even if a particular application only needs a few functions from it. In practice, a large fraction of residual bloat resides in such libraries that static methods cannot trim. Thus, purely static attack-surface reduction has inherent flexibility and completeness issues.

*Major in computer science, minors in mathematics and physics

Author's Contact Information: Himanshu Dongre, hdongr2@uic.edu, University of Illinois Chicago, Chicago, Illinois, USA.

Another line of defense against memory-based exploits is to make software less predictable rather than smaller—for example, Address Space Layout Randomization (ASLR) or finer-grained code re-randomization techniques. While ASLR and its variants can hinder certain exploits, they do not actually remove any code or reduce what is available to an attacker; they only randomize addresses. Well-equipped adversaries have shown the ability to bypass randomization through information leaks or just-in-time code reuse attacks. For instance, just-in-time return-oriented programming (JIT-ROP) and other advanced code-reuse techniques can dynamically discover code gadgets even in the presence of ASLR, especially if all code remains mapped in memory. Furthermore, attackers have developed exploits like position-independent ROP (PIROP) and data-oriented programming (DOP) that render many traditional mitigations less effective. In short, mere randomization or static removal is insufficient against the evolving threat landscape—what is needed is a way to limit the code that is actually present and reachable by the attacker at any given moment, ideally in a manner that can adjust to the program's needs on the fly.

In this report, we introduce uffd-monitor, a new approach that replaces static debloating with a dynamic, execution-aware mechanism for attack surface reduction. The core idea is to treat a process's code pages as a cache that can be constrained in size and actively managed at runtime. Rather than mapping an application's entire code segment into memory, we allow only a fixed budget of k code pages to be resident at any time. As the program runs and execution moves from one region of code to another, the set of resident code pages is adjusted in real-time—pages of code that are no longer needed are evicted (unmapped), and new pages are paged in on-demand when the CPU begins executing code in those areas. This creates a moving sliding window of code visibility in memory. At any given instant, the process "sees" only a small window of its code; the rest of the code, while still logically part of the program, is absent from memory and cannot be executed or even referenced until it is brought back into the window. By dynamically hiding unused code, we dramatically reduce the attack surface available to an attacker who manages to hijack control flow. Any attempt to execute outside the current window will cause a page fault, halting the attack in its tracks. Compared to static debloating, this approach is far more flexible—all code can still execute when needed (so functionality is preserved), but code that is not needed at the moment is made temporarily invisible. Notably, this time-based reduction addresses the limitation of static techniques by re-introducing code on the fly when it becomes necessary. In essence, uffd-monitor achieves the security benefits of aggressive debloating (very little code exposed at once) without the risk of permanently removing something that might be needed later.

Uffd-monitor is implemented as a userspace tool and ita main contributions are:

- Dynamic code debloating design: We present the design of a sliding window code page management system that enforces a strict budget of code pages in memory. To our knowledge, this is one of the first systems to apply demand paging principles to executable code for security purposes, creating a moving window of code availability at runtime.
- Uffd-monitor implementation: We detail a prototype implementation that operates transparently alongside the application process, requiring no special hardware or custom kernel patches. This demonstrates that even complex, legacy-rich binaries can be dynamically "debloated" at runtime using existing OS mechanisms.
- Performance evaluation: We evaluate how reducing the in-memory code footprint affects performance. We measure the runtime overhead introduced by on-demand paging of code.

The remainder of this report is organized as follows. In Section 2, we provide background on memory paging, page faults, the Linux `userfaultfd` mechanism, and the `madvise` system call that underpins our approach. Section 3 then details the design of the uffd-monitor system, including how we enforce the code page budget and handle page faults. Section 4 briefly presents our developed implementation of the uffd-monitor and how the

rubber meets the road. Section 5 discusses the performance evaluation, and Section 6 concludes the report along with discussing future work.

2 Background

In this section, we summarize concepts that form the backbone of uffd-monitor.

2.1 Memory Paging and Code Pages

Modern operating systems use virtual memory and paging to manage program memory. Demand paging ensures that pages of a program (whether code or data) are loaded into physical memory only when they are actually needed. In the context of executable code, when a program starts, the OS does not load the entire code segment into RAM at once. Instead, the program's code segment is typically mapped into the process's address space backed by the program's binary file on disk. As the CPU begins executing instructions, if it tries to execute an instruction on a page of the code segment that is not yet in RAM, a page fault occurs. The kernel's memory manager intercepts this fault and brings in the missing code page from the executable file (this is a major page fault, requiring disk I/O). Once loaded, the instruction can be executed and the program continues. This on-demand loading of code pages is a standard mechanism in Linux and other OSes. Notably, because code pages are file-backed (the backing store is the executable file itself or a shared library file), the kernel can evict those pages from memory at any time and later reload them from the file. If physical memory is needed elsewhere, the kernel may page out infrequently used code pages by simply discarding them (since a pristine copy exists on disk). In fact, it is more efficient for the OS to drop code pages and re-read them from the file when needed than to swap them out to a dedicated swap area. In normal operation, if an evicted code page is later needed again, a page fault will occur and the kernel will reload the latest copy of that page from the binary on disk. Thus, code paging is inherently supported by the OS: only the code required for execution needs to be resident in memory, and unused code pages can stay on disk. However, on typical systems with ample RAM, once code pages are loaded, they tend to remain in memory unless memory pressure forces eviction.

2.2 Page Faults

A page fault is the key event in demand paging. It occurs when a program attempts to access a virtual memory address that is valid but whose page is not currently loaded in RAM. The hardware (MMU) raises a fault, which traps into the OS kernel's page fault handler. The kernel then determines the nature of the fault. If the address is part of a valid mapped region but simply not in memory (a not-present fault), the kernel will locate the backing store for that page, allocate a free physical page frame, and load the content. In the case of an executable code page, the backing store is the program's binary file; the kernel performs a file read (often via the file system page cache) to fetch the code bits into memory. After loading the page, the kernel updates the page tables to map the virtual address to the new physical page and marks it present, then resumes the program at the faulting instruction. The whole process is transparent to the program, aside from a slight delay. If, on the other hand, the address is invalid (no mapping or a protection violation), the kernel will signal an error (e.g., send `SIGSEGV` to the process). Page faults thus serve as the mechanism to bring pages into memory on demand and to enforce memory protection. They are typically handled entirely by the kernel.

2.3 Linux `userfaultfd`

Linux's `userfaultfd` is a feature that enables handling page faults in user space. Introduced in Linux 4.3, `userfaultfd` provides a file-descriptor-based interface for a userspace monitor to receive notifications of page faults in another process (or in specific memory regions of its own process) and to resolve those faults manually. In essence, `userfaultfd` allows userspace processes to take control of memory page faults that would normally

be handled by the kernel. To use this feature, a monitoring thread or process first creates a `userfaultfd` object and registers one or more memory regions to be tracked. The kernel then marks those regions such that if a page fault occurs on them, the faulting thread will be paused and a message describing the fault is delivered to the `userfaultfd` (which the monitor can read). The monitor can then take appropriate action, such as supplying the missing page data, and wake up the faulting thread once the page is in place. `userfaultfd` effectively implements a form of userspace paging: the userspace handler can decide how and from where to fetch the page contents.

2.4 madvise and Page Eviction

`madvise()` is a system call in the Linux kernel that allows an application to give the kernel advice or instructions about how to manage certain memory regions. Of interest is the `MADV_DONTNEED` flag, which informs the kernel that the specified memory range is not needed by the process at the moment. The kernel's response to `MADV_DONTNEED` is to immediately free those pages from the process's address space, discarding their contents. Any subsequent access to an address in that range will behave as if the page was never loaded, triggering a page fault that causes the data to be brought in again from the backing store if available.

3 Design

Uffd-monitor achieves executable code reduction by instrumenting the target executable code with a constantly running thread that dynamically modifies the resident set size (RSS) of the target process. We call this thread started by uffd-monitor the *fault handler thread (FHT)*, the reason of which will be clear subsequently. We refer to the virtual memory area (VMA) where the code pages are loaded and live within the RSS of the target executable as the *code VMA*. The FHT ensures that at any time the code VMA contains exactly k pages, where k is a fixed budget that can be smaller than the target's total code footprint.

3.1 Monitoring the Code VMA

To explain how the FHT restricts the code VMA of the target without perturbing its execution, we begin with a special case of it that maintains exactly one code page in memory. Clearly, if the target's execution is not to be perturbed, then the one page in the code VMA must be the one that is being executed, and must be swapped out of the code VMA for the next page that is jumped to by the target's execution in its control flow. This is exactly what the FHT does. Let us say that the current page being executed is page P_1 and the next page that the target's execution is going to jump to is page P_2. Since the target's code is instrumented such that only P_1 is currently in memory, the attempted fetch from an instruction from P_2 triggers a page fault. This page fault is intercepted by the FHT, which serves P_2 from a *monitor* containing the full code that resides in an anonymous read-only code repository elsewhere in memory. Similarly, when the target's execution jumps from P_2 to P_3, P_2 is swapped out of memory for P_3 and this swap is triggered by the page fault for P_3, which is handled by the FHT.

This mechanism can be generalized to the scenario in which the uffd-monitor maintains exactly k pages in memory. When a page fault is triggered, the uffd-monitor evicts the least recently used (LRU) page from memory and serves the code page from which the new instruction is being fetched.

3.2 Serving Code Pages

Uffd-monitor uses two kernel interfaces, `madvise` and `userfaultfd`, to monitor and transform the target. `madvise` is a system call in the Linux kernel that allows userspace processes to communicate the anticipated memory access patterns to the kernel. It allows uffd-monitor to notify the kernel to free particular code pages from the target's code VMA. Uffd-monitor calls `madvise` when it wants to swap out a page from memory for an incoming page. `userfaultfd` is a Linux kernel mechanism that allows delegating the handling of page faults for a memory region to userspace. When accesses to a region of memory attached to a `userfaultfd` file descriptor cause a

page fault, the kernel sends a request to a process reading from the descriptor. The process can then respond with the data for the page by writing a response to the file descriptor.

Upon startup, uffd-monitor registers the target's code VMA with a `userfaultfd` file descriptor and starts the FHT that polls on this file descriptor. When a page fault occurs, the FHT gets notified and not only evicts a page from the code VMA according to the LRU policy if the code VMA exceeds the k-page budget, but also serves a code page from the monitor that uffd-monitor sets up on startup.

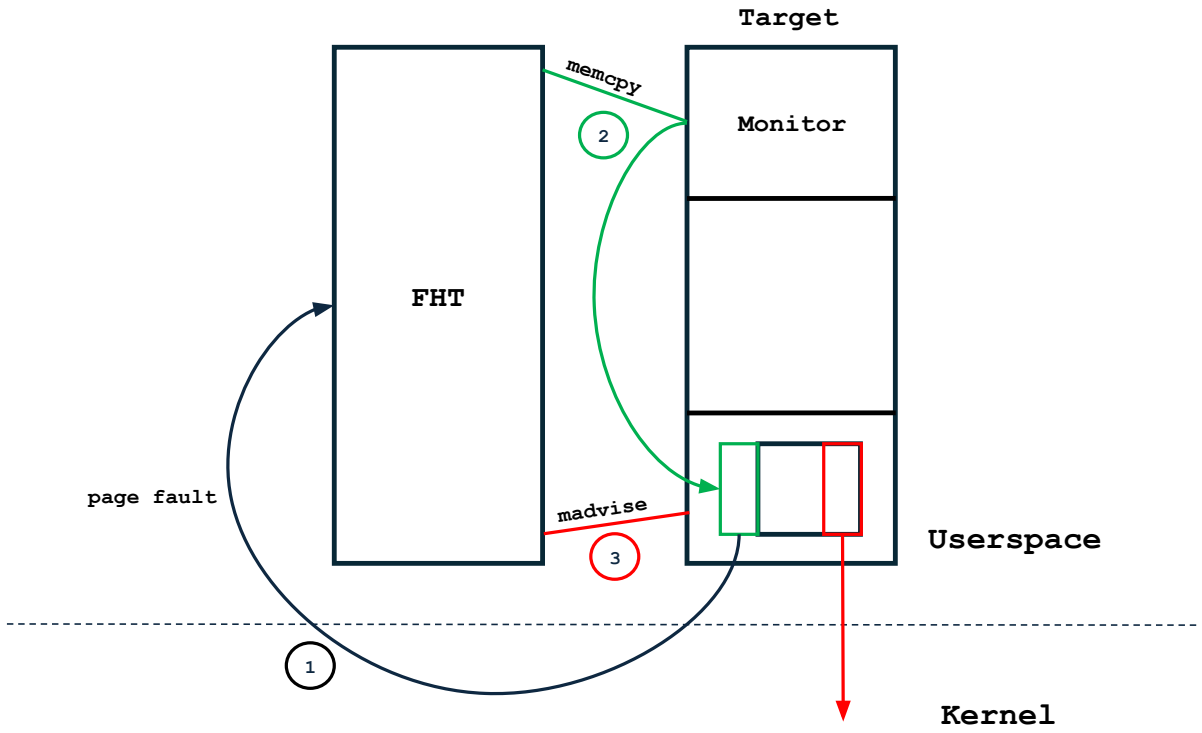

Fig. 1. Uffd-monitor's working

3.3 Sliding Window

In our design, uffd-monitor enforces a fixed code page budget by treating each code page's residency as a binary *visibility state*. Whenever the monitor handles a page fault and serves a code page into the target's code VMA, that page transitions from the *invisible* to the *visible* state, making its instructions available for execution. Conversely, when the monitor issues a `madvise` call to evict a page, that page becomes *invisible* from *visible*, effectively removing its code from the process's address space. Thus, the act of serving and evicting code pages can be understood as toggling their visibility, and, in turn, as an act of toggling the code visibility of the target process. Under this notion of visibility, the k-budget enforced by uffd-monitor can be understood as a constraint of keeping exactly k code pages visible in the code VMA at any point during runtime. In other words, the constraint can be formulated as maintaining a set of k visible pages whose contents change by one page after every page-fault-driven load paired with an eviction. We therefore characterize this set of visible pages as a **sliding window** of size k over the target's code VMA since the window constrains the target's code visibility to k pages keeping the

others invisible. The sliding window slides by one code page after every page-fault-driven load paired with an eviction.

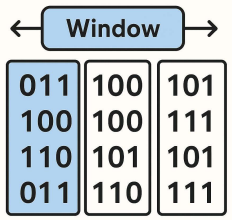

Fig. 2. A sliding window with a size of 1 page imposed by uffd-monitor on a target process with 3 code pages in total

Thus, the budget, k, can be understood as the window size. The sliding window notion provides intuitive clarity of how uffd-monitor contributes to dynamic attack surface reduction and code debloating without disrupting the control flow of the target process. The dynamism in the approach of uffd-monitor is captured in the sliding window notion by the ability of the window to dynamically slide over the target's code VMA.

4 Implementation

An implementation of the uffd-monitor design was developed on a x86-64 Ubuntu 22.04 Linux 5.15 with an 8-core Intel Xeon D-1548 2GHz CPU. In our implementation, we exploit the `LD_PRELOAD` facility provided by the GNU dynamic linker. `LD_PRELOAD` is an environment variable which specifies a colon-separated list of user-supplied shared objects that the dynamic linker must map into the process address space prior to any dependencies. We compile our uffd-monitor as a shared object and by leveraging this mechanism, load it into the target's virtual address space.

On startup, uffd-monitor reads the memory address mapping of the target's code VMA from the `/proc` directory to determine the memory addresses at which the code VMA starts and ends, the size of the code VMA, and the number of code pages it would contain at full capacity, which is the total number of code pages in the target's binary. Uffd-monitor uses this information to:

(1) copy the contents of the code VMA into a separate anonymous read-only VMA,
(2) remap the target's code VMA as an anonymous VMA and register it with a `userfaultfd` file descriptor,
(3) drop all pages from the target's code VMA using `madvise`, and
(4) start the FHT that polls on the `userfaultfd` descriptor,

before passing control to the target application. The sliding window size, k, can be set via an environment variable, `UFFD_MONITOR_SIZE`. The sliding window of size k is modeled by uffd-monitor as a FIFO queue of size at most k pages. When a page fault is triggered, uffd-monitor computes the offset of the faulting page address from the start of the code VMA and serves the code page at that offset in the monitor.

5 Evaluation

The on-demand eviction and reload of code pages via page-fault handling in uffd-monitor incurs non-negligible overhead compared to native execution. We evaluated the performance of uffd-monitor using benchmarks from the SPEC CPU2017 suite, some representative C applications, and the lighttpd web server,an I/O bound and event-driven server that serves static content, under sequential request load.

We plotted the wall-clock runtime (or time per request) of applications when running under uffd-monitor with a fixed code page budget k versus the code page budget k as a percentage of the total number of code pages.

The observed overhead is highly sensitive to two factors:

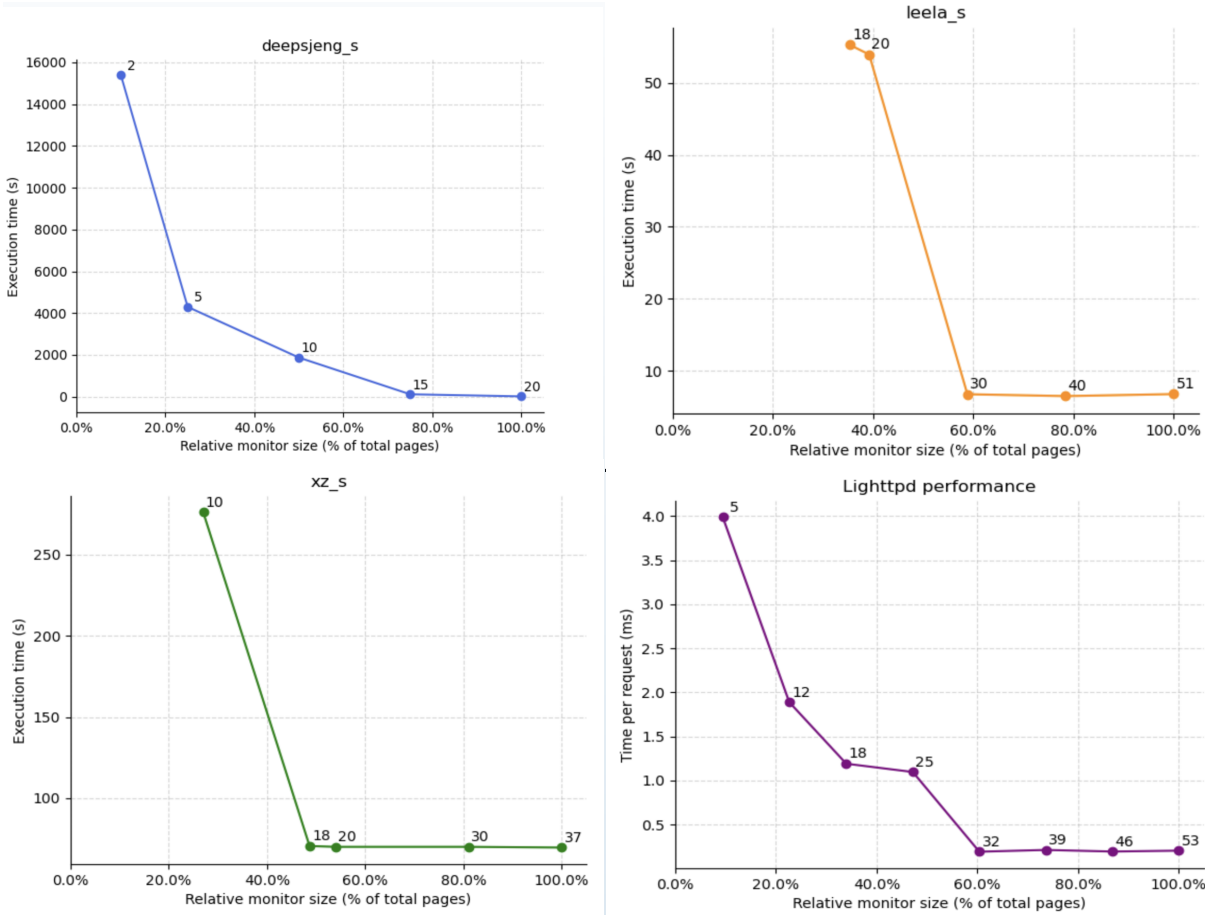

Fig. 3. Execution time vs. k-budget as a percentage of total code pages

Page-Access Working Set (r): Let the total number of code pages in the binary be t. At runtime, many programs execute most of their instructions within a small subset of pages, which we denote $r \ll t$. For example, a tight processing loop may repeatedly touch only pages 1 through r, while the remaining $t - r$ pages are invoked infrequently (e.g., error paths, initialization code, or cleanup code).

Budget k Relative to r: If the monitor's budget k satisfies $k \geq r$, then the entire working set stays resident, resulting in page-fault rates similar to an unmodified program under normal OS paging. However, setting $k < r$ forces constant eviction and reload within the hot loop, causing each pass through the loop to trigger $\approx r - k$ page faults per iteration.

Concretely, consider a loop that sequentially executes instructions on pages $1, 2, \ldots, r$ and then back to page 1. If $k = r - 1$, each time execution moves from page r back to page 1, page 1 must be faulted back into memory—yielding one extra major page fault per loop iteration. Since each major fault carries on the order of tens to hundreds of microseconds of latency, even a modest loop can see execution time explode. This behavior explains the pronounced "cliff" in slowdown observed for benchmarks such as `leela` and `xz` when k crosses the critical threshold at $k = r$.

Across our full suite of tests, we find:

- **Very low budgets** ($k \ll r$) lead to overheads exceeding 500%, rendering the approach impractical for performance-sensitive code.
- **Moderate budgets** ($k \approx 0.4\,t$ to $0.6\,t$) strike a balance: they still evict most cold code pages but keep the hot working set resident, resulting in average slowdowns in the range of 10%–30%.
- **High budgets** ($k \to t$) reproduce close to native performance but expose nearly the entire code footprint, diminishing security and code debloating benefits of uffd-monitor.

Thus, while uffd-monitor incurs variable overhead depending on an application's code page working set and the chosen budget k, our experiments indicate that selecting k at roughly half the total code-page count achieves a *practical* compromise: substantial reduction in instantaneous attack surface at the cost of modest performance degradation acceptable for many security-critical deployments.

6 Conclusion and Future Work

This report presents a novel dynamic runtime modification and code debloating tool called uffd-monitor. This tool restricts the in-memory code footprint of a target application to at most k pages, where k can be less than the total number of code pages in the target's executable. A developed implementation of the tool demonstrates the core working principle but also leaves spacious room for improvement, which will be the subject of our future work.

6.1 Policy for Serving and Evicting Pages

The evaluation naturally motivates further enhancements to reduce the performance overhead of uffd-monitor. In particular, we plan to design and implement a *predictive page-reloading policy* that proactively preloads the next code page(s) into memory before execution reaches them, rather than relying solely on page faults to trigger reloads. By anticipating control-flow transitions and issuing speculative load requests, this approach aims to smooth out fault-handling latency and further narrow the gap between monitored and native performance.

6.2 The Multi-Process Case

Some target applications fork child processes which could bloat the code. Uffd-monitor handles such targets by attaching the code VMAs of all child processes forked with their individual `userfaultfd` file descriptors. These `userfaultfd` descriptors are communicated to the parent process through a Unix domain socket (UDS), and the FHT in the parent process polls these file descriptors along with the `userfaultfd` descriptor of the parent process. When a page fault occurs on any one of the children's file descriptors, the faulting address is read, and an `madvise` system call is injected in the corresponding child process' context. Uffd-monitor uses `compel`, a library that facilitates implanting *parasites* into applications. Parasites are small binary blobs which execute code in the context of the target application.

Our implementation includes a handler for the multi-process case up to a single generation of child processes created via `fork()`. In this mode, uffd-monitor instruments and manages code pages for the parent and its immediate children. However, we have identified several bugs in the page-fault handling and synchronization logic for these child processes. Future work will focus on fixing these issues and generalizing support to arbitrary process hierarchies.

6.3 Further Security Hardening

Uffd-monitor serves the purpose of reducing the attack surface by minimizing the resident executable code footprint of a target application process. Uffd-monitor also involves, as part of its working principle, a trace of

code page accesses, which thwarts code reuse attacks. These security features of uffd-monitor can be combined with tracing system calls or data page accesses to provide a more security-enhanced software runtime.

Acknowledgments

To Professor Xiaoguang Wang, my research supervisor and Honors Capstone supervisor, for suggesting the idea, guiding me through the project, and tolerating my tardiness.

References

[1] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 353–362.

[2] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. 2020. Dynamic and secure memory transformation in userspace. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*. Springer, 237–256.

[3] Abhijit Mahurkar, Xiaoguang Wang, Hang Zhang, and Binoy Ravindran. 2023. Dynacut: A framework for dynamic and adaptive program customization. In *Proceedings of the 24th International Middleware Conference*. 275–287.

[4] Chris Porter, Sharjeel Khan, and Santosh Pande. 2023. Decker: Attack surface reduction via on-demand code mapping. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 192–206.

[5] Xiaoqi Song, Wenjie Lv, Haipeng Qu, and Lingyun Ying. 2023. LoadLord: Loading on the Fly to Defend Against Code-Reuse Attacks. *arXiv preprint arXiv:2303.12612* (2023).