DynaCut: A Framework for Dynamic and Adaptive Program Customization

Abhijit Mahurkar[*]
Virginia Tech
Blacksburg, USA
abhijitm@vt.edu

Xiaoguang Wang[*]
University of Illinois Chicago
Chicago, USA
xgwang9@uic.edu

Hang Zhang[†]
Indiana University Bloomington
Bloomington, USA
hz64@iu.edu

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

ABSTRACT

Software is becoming increasingly complex and feature-rich, yet only part of any given codebase is frequently used. Existing software customization and debloating approaches target static binaries, focusing on *feature discovery*, *control-flow analysis*, and *binary rewriting*. As a result, the customized program binary has a smaller attack surface as well as less available functionality. This means that once a software's use scenario changes, the customized binary may not be usable.

This paper presents DynaCut, for *dynamic software code customization*. DynaCut can disable "not being used" code features during software runtime and re-enable them when required again. DynaCut works at the binary level; no source code is needed. To achieve its goal, DynaCut includes a *dynamic process rewriting* technique that seamlessly and transparently updates the image of a running process, with specific code features blocked or re-enabled. To help identify potentially unused code, DynaCut employs an execution trace-based differential analysis to pinpoint the code related to specific software features, which can be dynamically turned on/off based on user configuration. We also develop automatic methods to locate code that is only temporally used (*e.g.*, initialization code), which can be dropped in a timely manner (*e.g.*, after the initialization phase).

We prototype DynaCut and evaluate it using 3 widely used server applications and the *SPECint2017_speed* benchmark suite. The result shows that, compared to existing static binary customization approaches, DynaCut removes an additional 10% of code on average and up to 56% of temporally executed code due to the dynamic code customization.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Systems security**.

KEYWORDS

Software Customization, Process Rewriting, Dynamic Attack Surface Reduction, Software Security

ACM Reference Format:
Abhijit Mahurkar, Xiaoguang Wang, Hang Zhang, and Binoy Ravindran. 2023. DynaCut: A Framework for Dynamic and Adaptive Program Customization. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3590140.3629121

1 INTRODUCTION

Software is increasingly becoming larger in code size and complex in functionality. According to recent research and online reports, more than half of the functionality in many software systems is rarely used [13, 21, 52]. A study on an industrial business information system reveals that more than 25% of its code has never been used for years [21]. Such unused components not only burden the code maintainers but also expose potential attack surfaces [21, 44, 45, 50].

In addition to code blocks that go unused for years, there exists code blocks with only *temporal liveliness*. For example, the initialization code of a long-running server program only executes during the boot-up phase. Since such code usually has access to sensitive system calls and configuration files, a good security practice is to remove them from memory after initialization [24]. However, this is often not the case in many server programs. Similarly, in some software systems, a good security practice would be to use certain features only under certain circumstances. For example, in a search engine database, it would be desirable to keep the data read-only for searching services during peak load times, while re-indexing or updating them during idle times (*e.g.*, at midnight) [18, 29]. Keeping *all program functionality accessible all the time* increases attack opportunities.

This raises the question of *how software can be debloated so that only the necessary code is kept in memory at any given time*? Software developers can get rid of unused code with the aid of, for example, static code coverage measuring tools [5, 25] that can help identify unused features [32]. However, this is generally not possible for end-users, especially for customizing off-the-shelf software without source code. To address this problem, recent works have focused on removing unused code directly from program binaries. For example, a number of efforts use control-flow information [23, 44] and static

[*] A. Mahurkar and X. Wang made equal contributions to this work. Most of X. Wang's work was done while he was at Virginia Tech.
[†] Most of H. Zhang's work was done while he was at Georgia Tech.

Middleware '23, December 11–15, 2023, Bologna, Italy

program analysis [2, 45, 50] to locate unused code and remove them through static binary rewriting or binary recompilation [44, 45, 57, 58, 65]. As a result, existing approaches take a vanilla binary as input and generate a debloated binary.

Such code removal is a static and one-time effort. Once modified, the code no longer changes. Such static debloating is sub-optimal in many real-world scenarios, where code that needs to be kept and removed may change as the program executes. On the one hand, it can be risky if static debloating is aggressive since the removed code may be required later. On the other hand, a conservative debloating strategy (*e.g.,* only exclude dead code [34]) may fail to maximize security benefits as it may retain code that is not used for a long time. This dilemma stems from the fundamental limitation of binary-oriented static debloating strategies, *i.e.,* they ignore the program execution time dimension, and are unable to dynamically and seamlessly remove and re-enable code.

Motivated by these concerns, we develop DynaCut, a dynamic and adaptive code customization system. DynaCut can disable and re-enable code paths of a process at run-time without interrupting its execution. This capability enables DynaCut to perform *dynamic code customization*: at any time during program execution, the required and undesired code can be individually configured (*e.g.,* initialization code can be kept during the setup phase and discarded later), maximizing debloating's security benefits.

DynaCut uses a *process rewriting* technique to modify a process at run-time. More specifically, DynaCut can efficiently snapshot a running process at any time and then transparently resume it with an updated snapshot (*e.g.,* with undesired code pages or basic blocks removed). This mechanism enables flexible dynamic debloating: different code blocks can be enabled/disabled during different time windows. While the required/unnecessary code can be identified either manually or by existing tools [23, 44, 50], DynaCut also contains a component to extract coverage information from execution traces and locate code blocks related to different software features, which can then be dynamically enabled/disabled. DynaCut can also help to automatically identify the temporally unused code (*e.g.,* initialization- or termination-related code) using the sequential execution order of code blocks and disable them, for example, after the initialization phase. This allows DynaCut to be used as a dynamic program debloating tool out of the box.

We implement a prototype of DynaCut and evaluate it using ten real-world applications, including widely used and security-critical web servers. The results show that DynaCut can identify and debloat up to 56% of executed code that is only used during initialization and can keep less than 17% code blocks visible in memory due to execution phase-based code customization. When dynamically customizing code features, the service interruption time is only ≈*400 ms*, resulting in no observable overall performance overhead. Our security analysis shows that DynaCut can mitigate several CVEs and known attacks including *CVE-2021-32625*, *CVE-2021-29477* and BROP attack [9]. To the best of our knowledge, DynaCut is the first dynamic code customization framework.

The paper's contributions include:

- We propose the concept of dynamic code debloating, which provides stronger security protection and flexibility over existing debloating approaches.
- We design and implement DynaCut, a first-of-its-class system for dynamic and seamless code customization. DynaCut is open-sourced.[1]
- We evaluate DynaCut's effectiveness and efficiency using real-world applications and benchmark suites and show that they can be dynamically customized with minimal service interruption.

The rest of this paper is organized as follows: Section 2 provides background information on code customization and describes the motivation. We then describe the design and implementation of DynaCut in Section 3. The evaluation is presented in Section 4. We discuss the future work in Section 5. Afterward, we summarize the related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

Software customization is a technique to selectively disable or enable software features [33]. Its primary applications include reducing the code size for easy software distribution and minimizing the program attack surface [23, 24, 30, 44, 46]. To minimize the software runtime attack surface, an ideal intelligent machine may control a tiny sliding window for code execution. In an extreme case, only one correct instruction is executable (visible) in the memory (Figure 1 (a)). Therefore, an attacker cannot arbitrarily jump to any vulnerable code that is marked as invisible, nor can he leaks any information about the code layout. However, such a machine does not exist for now. To make it more realistic, a number of efforts have focused on software customization from different perspectives. Feature-based software customization removes unnecessary code features using user-defined policies or program analysis results [33, 45, 46]. For example, a program's dependency graph can be obtained and embedded into the binary for customized program loading [45]. User inputs, specifications that distinguish unused features, and execution traces can be used to identify bloated code paths [23, 30, 44]. After retrieving feature-related code paths, many efforts re-assemble the binary to permanently eliminate unused code [57, 58] (Figure 1 (b)). Therefore, the resulting binary will be unusable in other scenarios that require the removed features.

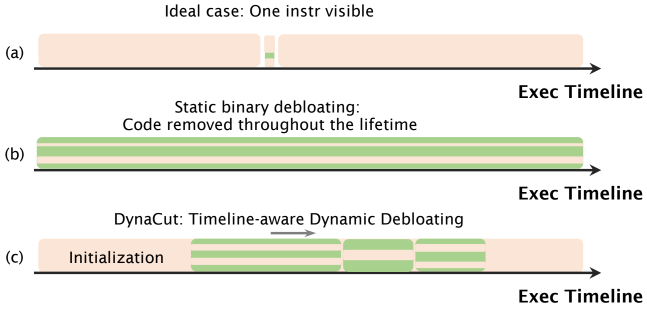
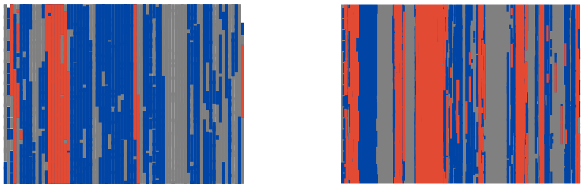

Ideal case: One instr visible
(a)
Exec Timeline
Static binary debloating:
Code removed throughout the lifetime
(b)
Exec Timeline
DynaCut: Timeline-aware Dynamic Debloating
(c)
Initialization
Exec Timeline

Figure 1: Software debloating for a reduced attack surface.

To demonstrate the existence of code bloating and better understand the size and distribution of unused code blocks across a process's lifetime, we analyzed the basic block liveness of a compute-intensive program (*i.e., 605.mcf_s* in SPEC CPU2017) and a server

[1] https://github.com/ssrg-vt/DynaCut



(a) *605.mcf_s* (b) *Lighttpd*

Figure 2: Visualization of process memory footprints for executed basic blocks (blue and red), unused basic blocks (gray), and initialization-related basic blocks (red) in *SPEC INT2017 605.mcf_s* benchmark and *Lighttpd* web server.

application (*i.e., Lighttpd*) and visualize the results in Figure 2. As can be seen, a significant percentage of basic blocks (in gray) has never been executed, showing the necessity and practicality of software debloating. Even though traditional static debloating techniques can eliminate the never-used code blocks (in gray), it cannot remove, for example, the initialization code blocks (in red) when they are no longer needed after the relatively short initialization phase. Given the considerable amount of such temporally alive code blocks, we argue that temporally removing them can further reduce the attack surface and increase security benefits.

Threat Model: We assume the attacker has remote access to the target process through a standard I/O interface, specifically, a socket connection. The attacker may also have access to the target binaries, such as the application and its libraries. DynaCut does not bring any exploit mitigations, but only reduces the attack surfaces through (dynamic) code customization and debloating. We assume the implementation of the disassembler is correct and sound; we also assume a strong trusted computing base (TCB), including the operating system kernel and the ELF loader. Side-channel attacks and kernel vulnerability exploits and mitigations are out of the scope of this paper.

3 SYSTEM DESIGN AND IMPLEMENTATION

DynaCut aims to dynamically customize code features of a process without interrupting its execution. To achieve this, DynaCut leverages a process rewriting technique to remove unwanted code blocks and transform the process's memory. Process rewriting can also re-customize the code features for the target process when the application scenario changes. For example, it can restore the removed code blocks and re-customize the code for a new application scenario.

At its heart, DynaCut is a *dynamic code customization mechanism* independent of the techniques determining what code to disable or enable. DynaCut can use existing binary analysis and debloating tools [23, 44] to find feature-related code paths. For example, control-flow trimming [23] analyzes execution traces and identifies the program Control-Flow Graph (CFG) edges that can be trimmed. Razor [44] similarly determines the desired code paths using execution trace logs. However, existing tools do not identify initialization code or do not support multi-threading applications. Therefore, we extend tracing-based code coverage techniques in DynaCut.

Figure 3 illustrates DynaCut. DynaCut has two major components: an *undesired code block identifier* and a *process rewriter*. The first component collects basic blocks from different execution traces and generates code coverage graphs. DynaCut uses trace log merging and code coverage graph comparison (*i.e., diff*) to determine wanted/undesired code blocks. The feature customization is based on the fact that most server programs handle different requests (features) using a big `switch-case` statement. DynaCut simply needs to locate the code dispatcher and cut the control flow edge to undesired features. DynaCut can also identify undesired initialization code that would not be executed post-initialization by analyzing execution logs.

Once the list of undesired code blocks is determined, it is input to the process rewriter. The process rewriter dynamically customizes the process based on the given code block list and a customization policy. For example, we can block the undesired features by replacing their corresponding code blocks (or only the first byte of each basic block) with the `int3` instruction[2]. We can also unmap corresponding code pages if the undesired feature consists of a large memory footprint. Subsequently, when undesired code blocks are executed, an exception will be raised. The process rewriter can further customize the program behavior when the exception is raised. For example, it can inject signal handlers into the target process's address space and update the process's default signal handlers. Users can also specify policies, such as terminating program execution or safely skipping undesired requests. DynaCut also allows users to restore the removed features by replacing the `int3` instructions with the original instruction bytes. All changes to the process are applied during the process's runtime. Therefore, even if the application scenario changes, end users can instantly update available features without restarting the process.

3.1 Undesired Code Block Identification

This component mainly uses execution traces to identify feature dispatchers and temporally undesired code.

Identify Feature-Related Code Blocks for Dynamic Customization: Similar to existing feature-oriented binary debloating techniques [23, 44], DynaCut also requires end users to specify the wanted and undesired features with sample inputs and record the corresponding execution traces to distinguish the undesired code blocks. The trace collector can either use a single trace file containing all the desired requests or merge multiple trace files of different requests. The undesired code identifier only needs execution traces of the basic block addresses and sizes (*i.e.,* tuples of <*BB addr, BB size*>) to differentiate the executions. The execution traces of undesired features are used to construct a code coverage graph $CovG_{undesired}$. Similarly, the wanted inputs can be used to construct a corresponding graph $CovG_{wanted}$. Since each graph contains a set of basic blocks, we can infer that the undesired code block blk satisfies the property: $blk \in CovG_{undesired}$ and $blk \notin CovG_{wanted}$. DynaCut narrows down the undesired code blocks

[2] The `int3` instruction is a one-byte breakpoint instruction in x86 CPUs. Upon executing `int3`, a breakpoint exception (#BP) is raised [31]. Other architectures have similar instructions for this purpose [53].

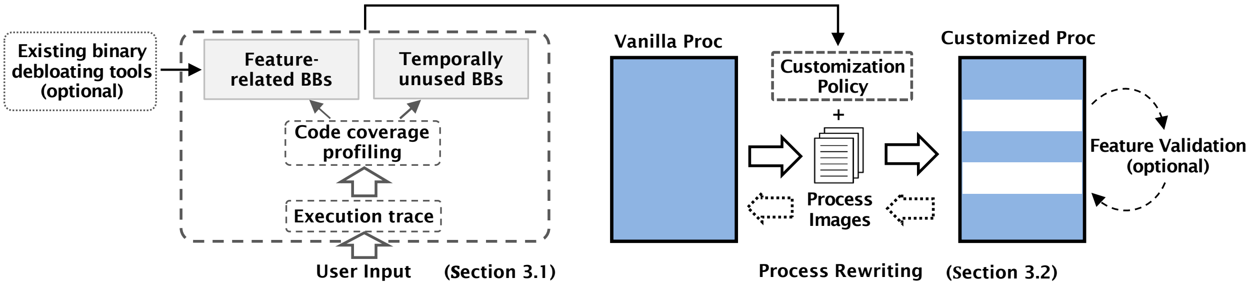
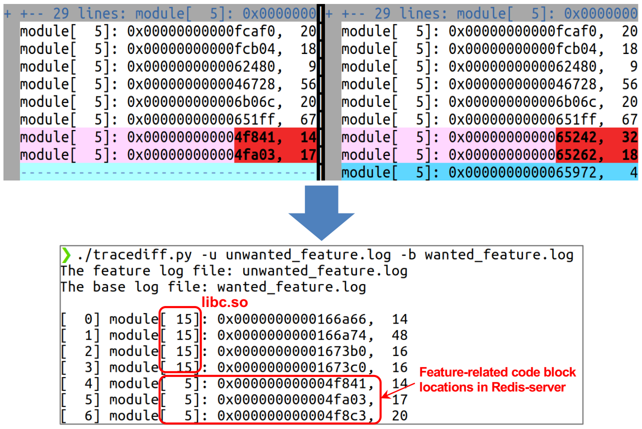


Figure 3: Overview of DynaCut. DynaCut consists of 1) feature-related code discovery (Section 3.1) and 2) runtime code feature customization with process writing (Section 3.2).

by filtering out basic blocks that appear in program libraries. Figure 4 shows an example of a feature-related basic block discovered from this process.



```
+ +-- 29 lines: module[  5]: 0x0000000 + +-- 29 lines: module[  5]: 0x0000000
module[  5]: 0x00000000000fcaf0,  20   module[  5]: 0x00000000000fcaf0,  20
module[  5]: 0x00000000000fcb04,  18   module[  5]: 0x00000000000fcb04,  18
module[  5]: 0x0000000000062480,   9   module[  5]: 0x0000000000062480,   9
module[  5]: 0x0000000000046728,  56   module[  5]: 0x0000000000046728,  56
module[  5]: 0x000000000006b06c,  20   module[  5]: 0x000000000006b06c,  20
module[  5]: 0x00000000000651ff,  67   module[  5]: 0x00000000000651ff,  67
module[  5]: 0x000000000004f841,  14   module[  5]: 0x0000000000065242,  32
module[  5]: 0x000000000004fa03,  17   module[  5]: 0x0000000000065262,  18
-----------------------------------   module[  5]: 0x0000000000065972,   4
```

```
❯ ./tracediff.py -u unwanted_feature.log -b wanted_feature.log
The feature log file: unwanted_feature.log
The base log file: wanted_feature.log

[  0] module[ 15]: 0x0000000000166a66,  14
[  1] module[ 15]: 0x0000000000166a74,  48
[  2] module[ 15]: 0x00000000001673b0,  16
[  3] module[ 15]: 0x00000000001673c0,  16
[  4] module[  5]: 0x000000000004f841,  14
[  5] module[  5]: 0x000000000004fa03,  17
[  6] module[  5]: 0x000000000004f8c3,  20
```

Figure 4: *Diff*-based feature-related basic block discovery: our *tracediff.py* tool automatically calculates undesired basic blocks using different execution traces.

The method of using basic block *diff*s from execution trace logs has a similar effect to that of using control-flow edges to identify unnecessary code blocks [23, 30, 44]. Both methods can find and remove undesired features (see Section 4). However, the code coverage *diff*-based approach is much easier to implement, as there is no need to reconstruct the CFG, especially for binaries without source code. We argue that our dynamic, code coverage-based approach is orthogonal to existing static program analysis approaches [23, 44]. They can be used together for accurate feature discovery.

Identify Temporally Undesired Basic Blocks: To identify the basic blocks of the temporally undesired code, we could attach timestamps for each executed basic block and assume that the blocks that completed before a particular timestamp as the initialization code. However, such an approach requires knowledge of code behaviors and manual analysis. For example, it's difficult to determine when the initialization code completes simply using the timestamps. Also, a basic block may execute during the initialization phase, and may also execute later. A similar problem is reported by Ghavamnia et al. [24]. In that work, the authors define a *transition point* between an application's initialization phase and its subsequent phase. For web server applications, they *manually analyzed the source code* to determine the transition points, such as Nginx's `ngx_worker_process_cycle()` function and Lighttpd's `server_main_loop()` function. For other applications, user annotations are expected for identifying transition points.

In DynaCut, we use a simple yet efficient approach to semi-automatically profile basic blocks only executed during the initialization phase. During code profiling, we ask end-users to notify the code coverage profiling tool that the target server program has initialized. The end of a program's initialization phase can be easily observed by *reading the printed log* or *using experience knowledge to wait a while* after launching the program. Upon receiving that signal, the tool dumps the execution trace collected so far, which is the code coverage of the initialization phase ($CovG_{init}$). The tool also clears the code cache and continues recording code execution. When the program finishes execution, the tool generates a second code coverage file containing the trace executed during the post-initialization phase ($CovG_{serving}$). From this, we infer the "not used" initialization code block blk that satisfies: $blk \in CovG_{init}$ and $blk \notin CovG_{serving}$. We also use the `tracediff.py` tool shown in Figure 4 to obtain an accurate list of the initialization basic blocks. Once the (temporal) undesired basic block list has been retrieved, we input it to the DynaCut runtime for dynamic code customization, together with the customization policy.

3.2 Dynamic Code Customization

DynaCut dynamically customizes a process without interrupting its execution. To achieve this, DynaCut relies on an online process rewriter. The process rewriter takes as input a process snapshot (*i.e.,* a process image), rewrites the snapshot by removing unwanted code, and then restores the process. End users can specify how undesired code should be removed. DynaCut allows users to simply block features or to fully delete them (*e.g.,* wipe code memory, unmap code pages). End users can also inject a customized exception handler for handling (unintended) undesired code access. For example, the handler can directly exit the program execution or return a customized error code but keep the program alive.

3.2.1 Process Rewriting. The process rewriter rewrites a static process image. To achieve this, DynaCut leverages CRIU [15], which is a userspace mechanism for checkpointing and restoring a running process for live process (or container) migration. DynaCut uses

CRIU to checkpoint a running process into a static process image. By rewriting a static process image, we avoid the complications of dealing with potential race conditions such as in a dynamic process transformation system [7, 39, 64].

The process rewriter supports *updating memory contents*, *increasing or unmapping the virtual memory areas (VMAs)*, and *inserting (or unloading) position-independent shared libraries into (from) the virtual memory space*. Updating the memory content allows replacing arbitrary instructions with one-byte `int3` instructions, thus removing small code features of a couple of bytes. We can also replace the first byte of an undesired basic block with an `int3` instruction, which blocks the code execution of an unwanted feature in a code dispatcher. Although this is sufficient to block that basic block from being executed[3], a powerful attacker may redirect the control flow to the middle of a basic block, launching an ROP attack [49]. To address this issue, the rewriter also allows an end user to wipe out a block of code memory or even unmap an entire memory page. This prevents access to any instructions of undesired code blocks. DynaCut also supports updating a process's exception handler and loading/unloading a shared library dynamically. The transformed process image can be safely restored using CRIU.

In summary, the DynaCut process rewriter can safely transform a live program to enhance the security of specific execution phases dynamically. The DynaCut process rewriting differs from dynamic binary instrumentation (DBI) [11, 38]. DBI tools such as DynamoRIO [11] and PIN [38] generate the code on-the-fly and use a code cache to store the translated code at runtime. In contrast, DynaCut statically updates the target code within a small time window. This prevents potential race conditions between the target process and the process rewriter. Moreover, once the process is restored, static process rewriting has almost zero runtime overhead, which is impossible using DBI tools. To support multi-process applications, DynaCut iterates through each process's memory space and updates the corresponding code.

3.2.2 *Block Undesired Features.*

Once a list of basic blocks has been identified as an undesired code feature during the analysis phase, DynaCut blocks the feature by placing an `int3` instruction in the first byte of the first basic block executed in this list. As mentioned earlier, basic blocks in the undesired feature list are *unique* code for that feature. Therefore, blocking the first instruction (byte) from being executed is enough to disable that feature. Alternatively, end-users can specify a more aggressive policy to entirely remove unwanted code by replacing basic blocks on that code path with `int3` instructions. The second policy increases security as it does not allow code reuse attacks on unwanted code. However, it adds performance overhead if the end-user wants to restore that feature.

DynaCut also allows an end-user to define how the program behaves when inadvertently accessing the blocked feature. Most existing binary debloating works terminate the program if users accidentally execute the blocked code feature [23, 44, 50], which brings usability issues. DynaCut addresses this problem by allowing end-users to program applications' behavior when accidentally accessing blocked features. Specifically, DynaCut allows inserting a signal handler to capture the unexpected `int3` execution (`SIGTRAP` exception). There are multiple strategies for the signal handler to deal with `SIGTRAP`. For example, users can call `exit()` to terminate execution, like most existing works do. For applications with *default error handling code*, users can program the behavior of accidental access to the blocked code.

When the blocked code is touched, the DynaCut-inserted signal handler can capture the exception and obtain the execution context. It then updates the instruction pointer by adding an offset value to the exception address so that upon signal return, the instruction pointer points to a new location where the application handles the wrong request. For example, when we disable `PUT` and `DELETE` methods (L5 and L8 in Listing 1) in a web server, we can program the fault handler to jump to the code that responds a `403 Forbidden` (L12 in Listing 1). Therefore, even if end-users inadvertently access a disabled method, they only receive a `403 Forbidden` response instead of terminating the web server.

```
1  static ngx_int_t
2  ngx_http_dav_handler(ngx_http_request_t *r)
3  {
4      switch (r->method) {
5      case NGX_HTTP_PUT:
6          ...
7          return NGX_DONE;
8      case NGX_HTTP_DELETE:
9          return ngx_http_dav_delete_handler(r);
10     ... ...
11     }
12     return NGX_DECLINED;
13 }
```

Listing 1: Code snippet of Nginx's request handler.

Figure 5 illustrates DynaCut's runtime code feature blocking capability using process rewriting. The updated memory is shown on the right of Figure 5. DynaCut allows multiple code features to be blocked by replacing the first byte with an `int3` instruction (machine code `0xCC`). When the unwanted code is inadvertently accessed, a `SIGTRAP` will be raised (steps ① and ②). The execution redirects to the fault handler, where we update the instruction pointer so that upon signal restoration, the application jumps to the code that responds with a `403 forbidden` message to the HTTP client (step ③). Thus, the web server's available features are dynamically blocked without interrupting the service. Similarly, end-users can restore the original instructions for those disabled features if the use scenario changes. By such a "bidirectional" process transformation, end-users can dynamically maintain a minimal available code feature set to reduce the attack surface.

Currently, we support programming the unintended behavior only if the target program *has an error handler*. We also require that the entries of the default error handler and unwanted code features reside *within the same function*. This is common for server applications as they often have a large *switch-case* statement to dispatch different client requests to their respective handlers. Thus, updating the instruction pointer with an offset does not mess up the function call stack. In the future, we expect to use program analysis and stack rewriting techniques to update the execution context if the entrances of the code handlers are in different functions. We terminate program execution for applications that do not have default error handlers, similar to most software debloating works [23, 44, 50].

[3] A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.

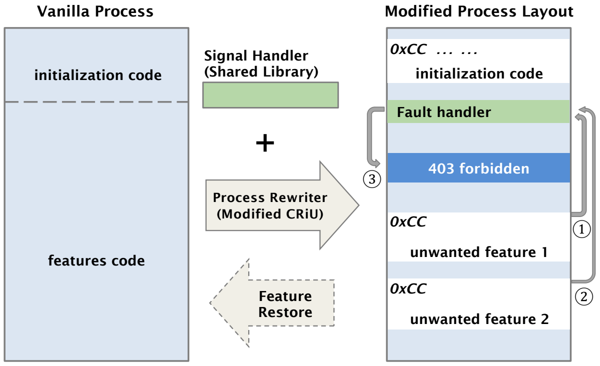


Figure 5: Illustration of DynaCut's code feature blocking and control flow redirection capabilities.

3.2.3 Validate Functionality for Removed Code. Similar to many existing binary debloating works [23, 30], DynaCut may also suffer from *over elimination* during the unused code identification phase. This is because the sample input for trace gathering may not be sufficient to cover all desired code paths, leading to some potential code paths being wrongly classified as unused. To validate the correctness of a customized process image, DynaCut allows end-users to inject a verifier library to check if any desired basic blocks have been falsely classified and removed as unintended code. This is similarly accomplished through the `SIGTRAP` handler. Instead of terminating program execution upon executing an unintended trap instruction, the verifier library restores the original instructions and logs the false addresses. This allows end-users to validate whether the functionality remains correct after code customization.

3.2.4 Other Code Customization Policies (Use Cases). Although DynaCut focuses on the mechanism for dynamic program customization, its flexible process transformation capabilities allow different customization policies. For example, the DynaCut users (or administrators) can dynamically expose the minimal functionalities of server programs according to the required workloads. Specifically, an administrator can disable any data write capabilities of web and database servers to prevent attackers from maliciously modifying data on an online system. Meanwhile, only re-enable the write capability when the administrator needs to update the content. This minimizes the time window for an attack.

Similarly, the administrator can use DynaCut to minimize potential vulnerabilities of new versions of software components. New software versions often likely contain *zero-day* bugs since they are less tested and deployed [63]. Also, many new features in new software versions may not be used by other legacy software components. With DynaCut, the administrator can disable unused new features and re-enable them only when these features are required. The feature customization is instant and will not interrupt the service. This reduces the attack window since the longer new features are used and tested, the fewer bugs they are likely to have.

3.3 Implementation

We implemented a prototype of DynaCut. Our implementation leverages CRIU [15] to checkpoint a running process and save its memory pages, register states, opened files, and network connections into several process images. After being transformed by DynaCut, the saved data will be used to restore the process exactly as it was during the checkpoint. CRIU is especially useful for transforming stateful programs with live connections, such as most web servers and key-value stores, as it supports `TCP_REPAIR`, which allows re-establishing saved TCP connections.

***Modifications to CRIU:*.** To implement DynaCut, we made several changes to CRIU. CRIU only dumps the anonymous pages of a process to a file. This saves network bandwidth for transmitting process image files during process migration. Code pages do not have to be saved because file-backed memory can be reconstructed by the page fault handler when a restored process attempts to access the virtual memory again. In DynaCut's implementation, we added an option in `criu/mem.c` to dump the private and executable pages (*i.e.,* `PROT_EXEC` and `FILE_PRIVATE`).

We also extended the CRIU image tool to support process rewriting. CRIU has an image checking tool called CRIT [16] that is used to examine process images in the protocol buffer format (protobuf) [26], decode them to human-readable JSON files (`decode`), and encode them back to the protobuf format (`encode`). For example, users can use CRIT to print all memory regions of the application (*i.e.,* `crit x <dir> mems`) or check the register values of a process snapshot (*i.e.,* `crit show core.img`). We made extensive changes to CRIT to provide easy-to-use APIs for process transformation. We added support to update memory contents, enlarge or unmap the VMAs, and insert position-independent shared libraries to the virtual memory space. Our CRIT extension also supports removing a single basic block/function given a base address, the size of the basic block/function, and its file offset.

DynaCut can dynamically load a customized exception handler. To accomplish this, DynaCut modifies the CRIU images and inserts the library into the process address space. In particular, DynaCut rewrites the following images:

The `core` *image file.* This file contains process information including binary name and location, signal handlers and masks, and register values, among others. DynaCut modifies this file to add the *signal handler address*, *restorer address*, and *signal mask* into the `SIGTRAP` sigaction field of the file. For the signal restorer, instead of using the default one present in the application, we add the restorer code from the code pages of the signal handler library itself. It is a 9-byte code that issues a `rt_sigreturn` syscall. The signal handler address is calculated by adding the file offset of the signal handling function with the VMA base address of the user's choice.

`pages`, `pagemap` *and* `mm` *image files.* The raw page contents are stored in the `pages.img` file, while the `pagemap.img` file contains information about which virtual memory regions are populated with data. To load a shared library into the target address space, we need to create new virtual memory regions and insert new memory pages that contain the library's code and data. DynaCut's process rewriter parses the shared library and calculates the size of each ELF section. This is very similar to a traditional ELF loader, but DynaCut loads the shared binary and dynamically injects it into running processes. DynaCut allows the end-user to specify where the shared library must be loaded. By default, DynaCut loads the shared library into a randomized but unused location. DynaCut further encodes the new pagemap information into the protobuf image. The mm file contains information about the application's VMA regions: their start addresses, end addresses, file offsets, shared memory IDs, permission flags, and status flags. The mm.img file differs from the pagemap.img file in that the pagemap.img file only contains details about pages that are *populated*, whereas the mm.img file is a collection of all the VMA regions of the application. DynaCut also modifies the mm.img file to update the VMA information, such as adding the start address, the end address, the file offset, and the corresponding permissions for the added shared library.

We leverage the pyelftools [6] to parse ELF libraries. Pyelftools reads raw data from the shared library ELF; we then add this data to the CRIU pages.img file by creating new pages and ordering them according to the pagemap.img file. End-users can specify any 64-bit userspace address that is not used by the process as the base address of the signal handler. DynaCut also performs global data relocations and procedure linkage table (PLT) relocations [36] with respect to the user-specified address. Global data relocations are performed by adding the VMA base address of the library to the st_value field of the symbol. For PLT relocations, we first find the external libc function symbol offset from the libc binary. Next, we add the runtime VMA base address of libc to these symbol offsets and write the new addresses to the global offset table (GOT) [36] of the signal handler library.

Trace Collection: DynaCut leverages a user specification to profile the application to generate different execution traces. DynaCut also requires the end user to generate as many use cases as possible for both wanted and unwanted features. In this regard, fuzzing techniques can partially help to achieve higher code coverage [66]. To collect code coverage logs, we run the target binary under DynamoRIO's drcov tool [19]. DynaCut provides a script to directly print feature-related code blocks from traces of wanted and unwanted features (Figure 4's tracediff.py). We extended DynamoRIO to enable dumping of the initialization phase's code coverage. We used DynamoRIO's communication mechanism called nudges [20], to dump the code coverage of the initialization phase. Our extended DynamoRIO tool dumps the rest of the code coverage when the program finishes.

A Prototype of DynaCut: We added 630 lines of C code and 2,696 lines of Python code to CRIU/CRIT for process rewriting. For implementing profiling of the initialization basic blocks, we added 108 lines of C code to DynamoRIO. We also developed scripts to automatically rewrite processes for given tasks, such as finding feature-related basic blocks and dynamically disabling code blocks. These scripts run externally to the target program. To reduce the time for storing a process image on the hard disk, we checkpoint the process images into an in-memory filesystem, *i.e.,* tmpfs [47].

4 EVALUATION

In evaluating DynaCut, our primary goals include understanding its runtime overhead and security benefits. We also aim to use DynaCut to reduce the attack surface of a real-world application by dynamic code customization. We demonstrate this using the Nginx web server.

Experimental setup. Our experiments were performed on a laptop with an Intel i5-10210U CPU (1.60GHz, 16GB RAM, Ubuntu 20.04 LTS with kernel version 5.8.0). To evaluate DynaCut's different functionality and features, we chose a diverse set of applications. We used the SPEC INT2017 benchmark suite as representative of CPU- and memory-intensive workloads. Since web servers are often security-, performance-, and reliability-critical (*e.g.,* low tolerance to service interruption), they pose high requirements for a dynamic code customization tool. We used two web servers, Nginx (v1.18.0) and Lighttpd (v1.4.59), as representative server applications. Nginx uses multiple processes, organized in a master-worker style. Lighttpd has an event-driven single-process architecture. Similar to web servers, in-memory key-value stores also have high security, performance, and reliability requirements. In addition, they have a well-defined feature set, which makes them suitable for evaluating DynaCut's feature removal functionality. We chose Redis (v6.2.3) as a representative key-value store.

4.1 Performance Overhead

We evaluate DynaCut's overhead by measuring how long it takes to rewrite a running program and the duration of the service interruption. DynaCut supports two types of code removal: feature removal and initialization code removal. Since the process rewriting policies are slightly different, we report the time costs separately. For both types of overhead measurement, we created a tmpfs for storing the intermediate process state. We measured the time cost using Linux's date command in nanosecond precision.

Feature removal overhead. For feature removal, DynaCut modifies CRIU images to disable feature-related code paths and loads a fault handler for handling unintended feature requests. Therefore, the overhead includes process dumping and restoring, instruction replacement, and loading the dynamic shared library code. We configured both web servers to use the web distributed authoring and versioning (WebDAV) extension [62]. We selected a few request types as potential unintended features. For example, we chose the PUT and DELETE requests in Nginx and Lighttpd as unintended requests, and chose the SET command as the unintended request in the Redis key-value server. We used these features as unintended features simply because they can potentially be used to alter read-only data in the servers.

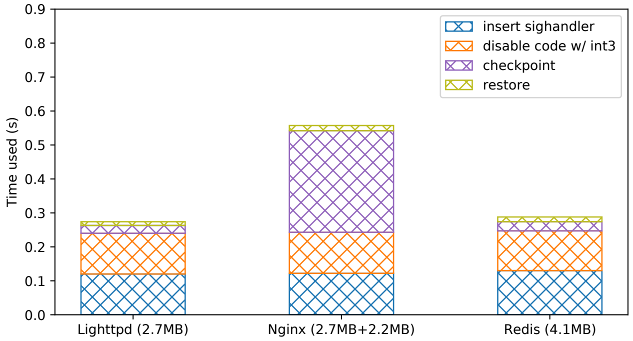
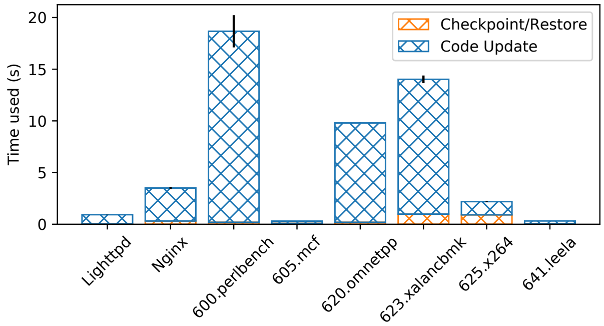


Figure 6: DynaCut's overhead for dynamically customizing code features.

As shown in Figure 6, DynaCut takes about 0.274 seconds and 0.56 seconds to customize the Lighttpd and Nginx instances, respectively, for feature customization. For Redis, the time taken is about 0.29 seconds. These are average numbers of repeating 10 times, with a standard deviation of 17 *ms*. The time taken for customizing the features of the applications are similar, but the checkpointing times are slightly different. For example, it takes 0.3 seconds to checkpoint Nginx, as Nginx has two processes to snapshot (2.7MB and 2.2MB are the sizes of each Nginx process image, as shown in Figure 6). For feature customization, DynaCut only needs to find the unintended code block by its address, replace the first byte of the feature-related basic block, and insert the fault handler. Thus, the time cost is almost constant.

Initialization code removal. As described before, initialization code removal replaces the code blocks that are used only during the initialization phase. Unlike feature removal, the overhead of initialization code removal is mainly due to replacing all unused basic block instructions. Figure 7 shows DynaCut's time taken for removing initialization functions. The sizes of the `.text` section and the CRIU process image for each application are shown in the table included in the figure. For Nginx and Lighttpd, the overhead incurred for modifying the images is about 3.5 seconds and 0.93 seconds, respectively. Most of this time is used to analyze the process images and remove the initialization code (*e.g.,* replace instructions with `int3` or unmap certain pages).



code size	335KB	853KB	1.96MB	18.36KB	1.56MB	4.6MB	570KB	189KB
image size	2.3MB	4.9MB	184MB	28MB	214MB	191MB	156MB	9.7MB

Figure 7: DynaCut's overhead for customizing initialization code in process images.

DynaCut's evaluation for the SPEC benchmarks is slightly different from that of the servers. The SPEC benchmarks are CPU/memory intensive, and unlike servers, they do not have a clear boundary between the initialization and serving phases. We chose the initialization point that we observed when the application was fully started. We used SPEC's INTSpeed suite and evaluated seven C/C++ benchmarks of the suite.[4] `605.mcf_s` is the smallest benchmark in the suite, and when compared to the other benchmarks, the overhead of modifying it, 0.22 seconds, was negligible (Figure 7). In contrast, `600.perlbench` is the most expensive case for initialization code removal, taking about 18 seconds.

The time for modifying the process images of the different benchmark programs depends on various factors, such as the initialization/serving transition point, the size of the CRIU images, and the number of initialization code blocks. Figure 7's graphs for `600.perlbench_s` and `623.xalancbmk_s` illustrate this. Even though `623.xalancbmk_s` has a larger `.text` section size and both programs have a comparable size for their image dumps (184MB vs. 191MB), the time taken to modify `600.perlbench_s`'s image is about 4 seconds more than `623.xalancbmk_s`'s. This is because, we chose an initialization point that is much deeper for `perlbench_s` than for `xalancbmk_s`, causing the extra overhead. The number of initialization basic blocks identified for removal also varies. For `perlbench_s`, we identified about 10,808 basic blocks that can be removed. However, we only identified 6,497 of the same kind for `xalancbmk_s`. The overhead incurred is almost proportional to the length of this list of basic blocks. This is also evident in the graph: since `perlbench_s` has about 60% more basic blocks to remove than `xalancbmk_s`, `perlbench_s` takes about 50% more time than `xalancbmk_s` to remove initialization basic blocks.

Note that the time taken to remove the initialization code is a ***one-time cost***; it does not add any overhead to actual software deployment. Instead, end-users can directly restore the "customized" process image, which can be even faster than launching the program from the start.[5]

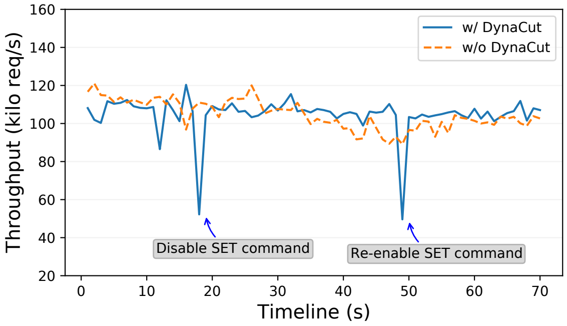


Figure 8: Redis server's throughput under DynaCut for modifying process images.

Service interruption time for dynamically customizing the Redis server. We measured the service interruption time when applying DynaCut during an application's execution. In this experiment, we set up a Redis server on our test machine and started a Redis benchmark instance (`redis-benchmark`) on the same machine, sending `GET` requests in an infinite loop. During the test, we dynamically applied DynaCut to the Redis server and rewrote the process to remove the code for handling the `SET` command, and later re-enabled it. We measured the throughput and latency and baselined them against an unmodified Redis server instance.

As shown in Figure 8, DynaCut does not terminate the Redis server. Instead, it only degrades the server's throughput within a

[4] We got an out-of-memory error when applying DynamoRIO's code coverage tool on `602.gcc_s` and `657.gx_s` benchmarks. We believe that DynaCut can customize the code using other code coverage tools.

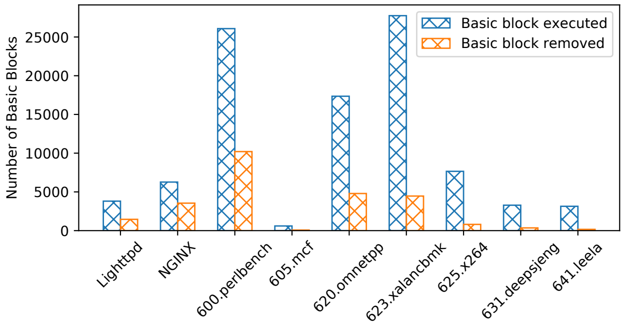
[5] Restoring a process image often takes a few hundred *ms* and is faster than running through the whole initialization process.

small time window of about one second. After the process rewriting, the customized process performs similar to the vanilla Redis server. We applied DynaCut again at the 48th second to re-enable the SET command. Both feature removal and re-enabling have similar performance costs. In this test, we did not trigger the SET request as it will fall through to the error-handling code, which will terminate the server's execution.

4.2 Security Evaluation

We evaluated DynaCut's security benefits by measuring the removed code block numbers and analyzing the attack surface reduction.

Number and size of code blocks removed. Unlike existing binary-based code customization approaches [23, 44, 50], DynaCut dynamically removes code blocks not used for current scenarios. For feature removal, the number of unused code blocks that DynaCut can disable heavily depends on the (undesired) feature selection. Recall that *DynaCut's main contribution is not to find the undesired features but to remove them dynamically.* Furthermore, existing binary debloating solutions are orthogonal to DynaCut in terms of feature removal. Therefore, we do not directly compare the code size reduction rate for feature removal with existing works. Instead, we report the number of initialization-related basic blocks removed (Figure 9) and show how many basic blocks live in the memory for each execution phase under DynaCut and compare them against previous works [30, 44] (Figure 10).



total BB #	17.8k	35.4k	139k	1180	115k	310k	22.2k	5026	10.6k
code size	335KB	853KB	1.96MB	18.36KB	1.56MB	4.6MB	570KB	81KB	189KB
init code rm	31.6KB	100KB	178KB	2.46KB	102KB	92KB	22.9KB	7.14KB	4KB

Figure 9: Number of executed basic blocks, number of basic blocks removed by DynaCut, and the size of initialization code removed.

Figure 9 shows the result of removing initialization code. The first bar of each application shows the total number of basic blocks executed and the second bar shows the number of initialization basic blocks removed. The total number of basic blocks executed is a deduplicated number calculated from the drcov trace. We also report the number of total basic blocks, the code size, and the size of initialization code removed in each binary. The number of total basic blocks of each binary is obtained using Angr [51]. As seen in Figure 9, DynaCut can remove up to 56% of the *executed basic blocks* in Nginx with an overhead of about 3.5 seconds (Figure 7). Similarly, DynaCut removes about 46% of the executed basic blocks in a Lighttpd process. Many executed code blocks can be removed mainly because web server applications spend an extensive amount of cycles loading their configuration files and initializing worker threads. Once the server applications are fully initialized, they usually execute an event loop to dispatch different client requests. Therefore, the "hot" code block numbers are relatively smaller than that of other applications. For SPEC benchmarks, DynaCut removes 8.4% to 41.4% of executed code blocks with an average of 22.3%. The highest percentage case in SPEC INTSpeed is `600.perlbench_s` with about 41.4% of the executed basic blocks identified as initialization code blocks and removed. Interestingly, this is a Perl application that processes email text and also executes in a loop. We also show the removed code size in Figure 9. For server applications, DynaCut removes about 10% of the unused initialization code in size.

Next, we use *Lighttpd* as an example to show the minimal amount of live code DynaCut can maintain over time. Here, "live" means code blocks that an attacker can reach. We mimic a scenario of using a web server to serve web pages most of the time and dynamically opening a time window for the system administration (*e.g.,* uploading files to the server). Figure 10 shows the result. The dashed lines indicate the percentage numbers of live basic blocks in different code customization techniques (*i.e.,* Razor [44] and Chisel [30]) and are normalized against the vanilla *Lighttpd* binary. The line of DynaCut shows the number of live basic blocks in each execution phase. After *Lighttpd* finishes initialization, the administrator sends DynaCut a command to remove the initialization code. When he needs uploading files, he can enable the HTTP PUT method. This allows him to manipulate files on the server (time slot 8-9 in Figure 10). Since DynaCut allows dynamically updating code liveness (in the aforementioned sense), it maintains a smaller amount of code for each phase. As a result, DynaCut keeps less than 17% of code blocks visible in memory during the lifetime of *Lighttpd*, better than the state-of-the-art binary debloating techniques [30, 44].

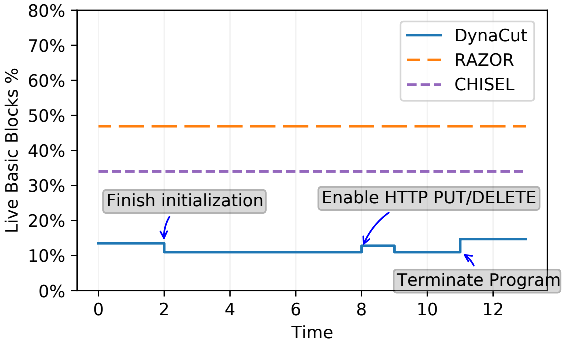


Figure 10: Number of live basic blocks over time.

We should note that the live basic blocks in each stage can be completely different, although the numbers are slightly different in each execution phase. Specifically, live basic blocks before timestamp 2 (*i.e.,* initialization-related) mostly differ from live basic blocks after timestamp 2 (*i.e.,* initialization code removed). Feature-related code can also be dynamically enabled when the workload changes. In contrast, existing binary-oriented debloating techniques cannot minimize code footprint [30, 44]. According to Qian et al. [44], Razor and Chisel remove an average of 53.1% and 66% basic blocks, respectively. Due to their static design, the amount of live code blocks is significantly larger than what DynaCut achieves (*i.e.,* a maximum of 17% code blocks visible in memory). In this experiment, we manually run scripts to rewrite the process for each execution phase. In future work, we plan to fully automate this.

Attack surface reduction. DynaCut can remove undesired code features on the fly; thus, those features cannot be maliciously executed whenever they are not in use. DynaCut can also be used in an alternative way to maintain a minimal set of needed features: developers can first disable new features in an upgraded software. When a new feature is required, DynaCut can dynamically enable that feature. This can be especially helpful in keeping the whole system secure when integrating new software versions into existing systems. New code features often contain zero-day bugs. For example, newer Redis versions support complex algorithms for string operations (*e.g.,* the STRALGO command). Such new commands may not be needed for legacy code developed using older Redis versions. Using DynaCut, software system maintainers can simply disable such not-in-use new commands until they are required. This allows legacy code to be protected from vulnerabilities introduced by new features (if these features are not in use). *CVE-2021-32625* and *CVE-2021-29477* are two such vulnerabilities found in recent Redis versions. We also examined Redis's other CVEs and confirmed that DynaCut could disable the vulnerable code paths (Table 1).

List of Redis CVEs [14] mitigatable using DynaCut	
CVE #	Description
CVE-2021-32625	STRALGO LCS command in Redis versions 6.0+ (integer overflow).
CVE-2021-29477	STRALGO LCS command in Redis versions 6.0+ (integer overflow).
CVE-2019-10193	SETRANGE command (stack-buffer overflow).
CVE-2019-10192	SETRANGE command (heap-buffer overflow).
CVE-2016-8339	CONFIG SET command in Redis 3.2.x prior to 3.2.4 (buffer overflow).

Table 1: Redis CVEs that could be mitigated using DynaCut's feature blocking capability.

We conducted another case study on DynaCut's security benefit obtained by removing the initialization code. Return-to-PLT (ret2plt) attack [48] is a variant of code reuse attack that invokes sensitive library code (*e.g.,* `execve()`) through exposed PLT entries. The procedure linkage table (PLT) is a small piece of trampoline code used to call external functions whose addresses are unknown at link-time. The PLT and the global offset table (GOT) provide application code with access to dynamically linked libraries. DynaCut wipes out initialization code, including the PLT entries used after initialization. In our evaluation, we found that DynaCut removes 43 out of 56 *executed PLT entries* in Nginx after the initialization phase is completed. After Nginx finishes the initialization, basic blocks that performed the `fork` were disabled because the worker process had already been created.[6] More importantly, the PLT entry for the libc `fork()` function was also disabled, preventing any ret2plt attacks that use the `fork()` function.

Blind ROP (BROP) [9] is a variation of ROP attacks that remotely locates ROP gadgets. It requires the server application to re-spawn crashed worker processes so that an attacker can brute-force the stack canary value. It also gathers and sends process information to the remote attacker through PLT entries like `write()`. DynaCut reduces the viability of attacks like BROP in two ways. First, DynaCut disables many executed but not-in-use PLT entries. Thus, finding a PLT entry for mounting the attack would be difficult. Second, DynaCut disables about 56% of the executed basic blocks, reducing the amount of available code for launching the attack. Even if the attacker can circumvent the disabled PLT entries and find enough gadgets, it would still be difficult to mount a BROP attack on the customized Nginx server. This is because, DynaCut also removes any code that can invoke `fork()` after initialization. If an attacker mounts a BROP attack on DynaCut-customized Nginx server, the first attempt to brute-force the stack canary will crash the worker process. We did a similar security evaluation for Lighttpd. Out of 57 total *executed* PLT entries, DynaCut was able to remove 33 of them. Some PLT entries that we disabled include `strcmp()`, `dlopen()`, and `socket()`. DynaCut's PLT entry removal sets it apart from existing debloating techniques. Existing techniques can remove unused code and, by extension, unused PLT entries, but DynaCut can remove *executed* PLT entries used only in particular execution phases.

5 DISCUSSION AND FUTURE WORK

The paper's main contribution is exploring the pros and cons of dynamic software feature customization by designing and implementing the DynaCut prototype. We acknowledge that DynaCut may suffer similar problems and challenges of using limited test inputs to precisely distinguish wanted and unwanted features, as many existing binary debloating systems have [23, 30, 44]. A complete and sound solution may require source-code level feature-code relationship analysis [22]. For example, we may improve DynaCut by automatically analyzing the source code to find each feature and the corresponding code blocks. We can then separate each feature-related code block into separate memory pages. As such, we can dynamically unload these code pages with DynaCut, faster than replacing code with `int3` instructions.

Combining program behavior with code debloating is another interesting future direction. For example, coarse-grained section-level binary information can be used to infer code intent [4]. Code usage under particular workloads can be machine-learned [37]. Moreover, we can monitor specific system calls to determine the end of the initialization phase, making DynaCut fully automatic. We leave these optimizations as our future works.

Currently, DynaCut only targets dynamically customizing features in application binaries. However, our approach can be extended to customize library code. There is a significant amount of

[6] Here, we configured Nginx to use only one worker process.

initialization code in the standard C library (*e.g.,* `glibc`) and other helper libraries (*e.g.,* `ld.so`). Some features in server applications are also loaded using shared libraries. We anticipate that unused shared library code can be dynamically unloaded through the process rewriting approach. Removing them from the process address space can further reduce the attack surface.

Lastly, we believe that *process rewriting* can be a general technique to solve other security and system problems, such as dynamically enabling/disabling *seccomp* filtering [24], live code rerandomization [64], dynamic software update [40, 42], and cross-architecture process migration and execution randomization [55, 60], among others. Process rewriting allows dynamically transforming the process state and memory layout from outside of the target process, preventing the transformation logic from being hijacked [64].

6 RELATED WORK

The first category of related work includes efforts on *binary debloating, analysis, and rewriting*. Binary debloating, which is closest to DynaCut, aims to reduce the binary size to reduce the attack surface. A number of debloating works focus on analyzing and removing unnecessary features [23, 30, 33, 44–46]. A major challenge is to accurately find control-flow transfer edges to unwanted code as over-identifying such edges can cause wanted features to be removed. Many recent works address this challenge from different perspectives [23, 30, 44]. For example, Razor [44] uses user-specified input cases and control-flow heuristics to ensure that all user-expected code blocks are removed. Chisel [30] applies reinforcement learning to build a statistical model that captures semantic dependencies between program elements and guides the search towards a desirable minimal program. Ghaffarinia and Hamlen [23] similarly apply a machine learning approach to execution traces generated from test suites to learn a subgraph of the developer-intended control flows. We argue that these approaches are orthogonal to DynaCut's central contribution, *i.e.,* a process rewriting mechanism to disable/enable code blocks from a process's memory layout at run-time. These prior works can therefore be used in DynaCut to infer a more accurate feature-related code path for dynamic customization.

After obtaining the feature-related control-flow transfer edges, many existing approaches use binary rewriting techniques to remove bloated code paths or simply block the related control-flow transfer edges [3, 12, 23, 44, 65]. For example, Uroboros [58] reassembles the disassembled code for program instrumentation. The reassembler recovers the semantic information from program assembly code and rearranges code and data on reassembling [57, 58]. However, program reassembling is a hard problem as compilers often discard linkage information. Even state-of-the-art reassemblers cannot achieve a 100% successful reassembly rate [57, 58]. BinRec [3] lifts a binary to an intermediate representation format, dynamically removes code features by allowing end-users to determine the required features based on a dependency graph, and then regenerates the target binary. However, binary recompilation cannot be directly used when feature requirements change during the lifetime of a program. DynaCut recognizes this, and enables dynamic code customization based on changing requirements. Other approaches use program analysis to find reachable code. BinTrimmer [46] uses value-flow domains to find and eliminate dead code. Quach et al. use a piece-wise compiler to embed the program dependency graph in a special section of the binary so that a piece-wise loader can directly load the needed code [45]. In contrast, DynaCut dynamically finds reachable code. Furthermore, DynaCut also allows a minimal amount of code features executable; the allow-list of features can then be gradually enlarged on demand.

The second category of related work includes efforts on *dynamically reducing the attack surface* [1, 24, 28, 35]. For example, Ghavamnia et al. [24] use static analysis to determine the syscall requirements for server applications after the initialization phase. Based on the analysis results, a customized `seccomp` filter is used to block unnecessary yet sensitive syscalls (*e.g.,* `execve()`, `fork()`) in the post-initialization phase. However, this approach still retains unused code in memory, creating potential attack opportunities through code reuse (*e.g.,* ROP). Shard [1] is a context-aware kernel specialization system that dynamically switches the execution context between a security-hardened kernel and a vanilla kernel. Face-Change [28] similarly profiles syscalls used by each application and changes the kernel view according to different application contexts. Compared to DynaCut, the dynamic kernel switching approaches are more heavy-weight. They require using a modified virtual machine monitor and also must recompile the target kernels.

Dynamic software patching (software repair) is another way to fix vulnerable code without stopping the process [27, 40, 42, 43]. Ginseng [40] uses a source-to-source compiler to generate update-able code and redirects function calls at runtime to make the updated code live. Kpatch [43] allows patching the Linux kernel without restarting or rebooting any processes using the `ftrace` mechanism. In contrast, DynaCut focuses on vulnerable code removal, but we believe similar dynamic software patching or software repairing systems can be built using DynaCut.

DynaCut also shares the ideas of *the principle of least privilege* and software fault isolation [8, 10, 17, 56, 67]. The principle of least privilege ensures that any entity of a computing system (*e.g.,* a process or a user) has access to only the necessary information for the intended functions [8]. Least privilege is often implemented using the concept of privilege separation, *i.e.,* splitting a system into different components with different levels of trust [10]. In practice, untrusted components are isolated into fault domains, preventing untrusted code from compromising the trusted computing base (TCB) [56]. Several efforts split complex software systems into multiple reduced-privilege compartments [10, 17, 67], and isolate different components of the application code [10, 41, 54, 56], untrusted third-party libraries [61, 67], or even different OS components [17, 59]. DynaCut dynamically updates the visibility of different code features and maintains the minimal code required for running software in a given scenario.

7 CONCLUSIONS

We presented DynaCut, a dynamic software customization system. DynaCut's key innovation is a novel process rewriting mechanism to update a process's state and memory layout at run-time. We built a prototype of DynaCut and used it to dynamically remove unused code features and temporally unused code from nine applications. Our evaluation shows that DynaCut dynamically removes up to 56% of *executed but unused* code blocks with ≈400 *ms* service interruption time. Compared to existing static binary debloating approaches, DynaCut minimizes the number of live code blocks in memory, further reducing the attack surface.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the US Office of Naval Research (ONR) under grants N00014-19-1-2493 and N00014-22-1-2672 and the US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

REFERENCES

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*.

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, David Balenson (Ed.). ACM, 70–83. https://doi.org/10.1145/3359789.3359823

[3] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 36, 16 pages. https://doi.org/10.1145/3342195.3387550

[4] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E Locasto, Jason Reeves, Sean W Smith, and Anna Shubina. 2013. ELFbac: using the loader format for intent-level semantics and fine-grained protection. *Dartmouth College Computer Science Technical Report* (2013).

[5] Ned Batchelder. 2021. Coverage.py. https://coverage.readthedocs.io/en/coverage-5.5.

[6] Eli Bendersky. 2022. pyelftools. https://github.com/eliben/pyelftools.

[7] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.

[8] Andrea Bittau. 2009. *Toward least-privilege isolation for software.* Ph. D. Dissertation. University College London, UK. http://discovery.ucl.ac.uk/18902/

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking Blind. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 227–242.

[10] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, Jon Crowcroft and Michael Dahlin (Eds.). USENIX Association, 309–322. http://www.usenix.org/events/nsdi08/tech/full_papers/bittau/bittau.pdf

[11] Derek Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, Taesoo Kim, Cliff Wang, and Dinghao Wu (Eds.). ACM, 23–29. https://doi.org/10.1145/3141235.3141243

[13] Mike Cohn. 2015. Are 64% of Features Really Rarely or Never Used? https://www.mountaingoatsoftware.com/blog/are-64-of-features-really-rarely-or-never-used.

[14] MITRE Corporation. 2022. Redislabs Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-18560/product_id-47087/Redislabs-Redis.html.

[15] CRIU. 2021. Checkpoint Restore in Userspace. https://criu.org/Main_Page.

[16] CRIU. 2021. CRIT: CRiu Image Tool. https://criu.org/CRIT.

[17] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 191–206.

[18] Sheng Di, Derrick Kondo, and Franck Cappello. 2013. Characterizing Cloud Applications on a Google Data Center. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*. IEEE Computer Society, 468–473. https://doi.org/10.1109/ICPP.2013.56

[19] DynamoRIO. 2021. DynamoRIO: Code Coverage Tool. https://dynamorio.org/page_drcov.html.

[20] DynamoRIO. 2021. Tool Event Model and API. https://dynamorio.org/using.html.

[21] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. 2012. How much does unused code matter for maintenance?. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 1102–1111. https://doi.org/10.1109/ICSE.2012.6227109

[22] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating Features in Source Code. *IEEE Trans. Software Eng.* 29, 3 (2003), 210–224. https://doi.org/10.1109/TSE.2003.1183929

[23] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1009–1022. https://doi.org/10.1145/3319535.3345665

[24] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[25] GNU. 2021. Using the GNU Compiler Collection (GCC): Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[26] Google. 2021. Protocol Buffers. https://developers.google.com/protocol-buffers.

[27] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

[28] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE Computer Society, 491–502. https://doi.org/10.1109/DSN.2014.52

[29] Alexander Halavais. 2017. *Search engine society.* John Wiley & Sons.

[30] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. https://doi.org/10.1145/3243734.3243838

[31] Intel 2018. *Intel 64 and IA-32 Architectures Software Developerś Manual.* Intel.

[32] Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 955–963. https://doi.org/10.1145/3338906.3340459

[33] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, Radu F. Babiceanu, Hélène Waeselynck, Raymond A. Paul, Bojan Cukic, and Jie Xu (Eds.). IEEE Computer Society, 122–131. https://doi.org/10.1109/HASE.2016.27

[34] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM Sigplan Notices* 29, 6 (1994), 147–158.

[35] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. 2011. Attack surface reduction for commodity OS kernels: trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11, April 10, 2011, Salzburg, Austria*, Engin Kirda and Steven Hand (Eds.). ACM, 6. https://doi.org/10.1145/1972551.1972557

[36] John R. Levine. 1999. *Linkers and Loaders.* Morgan Kaufmann, San Francisco, CA.

[37] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[38] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation.

In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 190–200. https://doi.org/10.1145/1065010.1065034

[39] 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 (Lecture Notes in Computer Science, Vol. 12308)*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider (Eds.). Springer, 237–256. https://doi.org/10.1007/978-3-030-58951-6_12

[40] Iulian Neamtiu, Michael W. Hicks, Gareth Paul Stoyle, and Manuel Oriol. 2006. Practical dynamic software updating for C. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 72–83. https://doi.org/10.1145/1133981.1133991

[41] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 241–254.

[42] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 573–585. https://doi.org/10.1145/3297858.3304063

[43] Josh Poimboeuf. 2014. Introducing kpatch: Dynamic Kernel Patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching

[44] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[45] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[46] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11543)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer, 482–501. https://doi.org/10.1007/978-3-030-22038-9_23

[47] Christoph Rohland. 2020. Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html

[48] Will Ryan. 2021. Buffer Overflows: ret2libc, ret2plt and rop. https://medium.com/cyber-unbound/buffer-overflows-ret2libc-ret2plt-and-rop-e2695c103c4c.

[49] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA).

[50] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 329–339. https://doi.org/10.1145/3238147.3238160

[51] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[52] Tom Taulli. 2019. Are Most Of Your Product's Features...Useless? https://www.forbes.com/sites/tomtaulli/2019/02/24/are-most-of-your-products-features-useless.

[53] Scott Tsai. 2021. Debug Break: Break into the debugger programmatically. https://github.com/scottt/debugbreak/blob/master/debugbreak.h.

[54] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[55] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M Tullsen. 2016. Hipstr: Heterogeneous-isa program state relocation. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 727–741.

[56] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, Andrew P. Black and Barbara Liskov (Eds.). ACM, 203–216. https://doi.org/10.1145/168619.168635

[57] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/ramblr-making-reassembly-great-again/

[58] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 627–642. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai

[59] Xiaoguang Wang, Yong Qi, Zhi Wang, Yue Chen, and Yajin Zhou. 2019. Design and Implementation of SecPod, A Framework for Virtualization-Based Security Systems. *IEEE Trans. Dependable Secur. Comput.* 16, 1 (2019), 44–57. https://doi.org/10.1109/TDSC.2017.2675991

[60] Xiaoguang Wang, SengMing Yeoh, Robert Lyerly, Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2020. A Framework for Software Diversification with {ISA} Heterogeneity. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 427–442.

[61] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and efficient in-process monitor (and library) protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security, EuroSec@EuroSys 2020, Heraklion, Greece, April 27, 2020*, Lorenzo Cavallaro and Andrea Lanzi (Eds.). ACM, 7–12. https://doi.org/10.1145/3380786.3391398

[62] Jim Whitehead. 2021. WebDAV Resources: Web-based Distributed Authoring and Versioning . http://www.webdav.org/.

[63] Wikipedia. Accessed: 2023-01-31. Zero-day (computing). https://en.wikipedia.org/wiki/Zero-day_(computing).

[64] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization.. In *OSDI*. 367–382.

[65] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[66] Michał Zalewski. 2021. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/.

[67] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.