DynaCut: A Framework for Dynamic Code Customization

Abhijit Mahurkar

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

August 12, 2021

Blacksburg, Virginia

Keywords: Software Debloating, Attack-Surface reduction, Return-Oriented-Programming, Process Snapshot, Dynamic Customization, Checkpoint Restore in Userspace

DynaCut: A Framework for Dynamic Code Customization

Abhijit Mahurkar

(ABSTRACT)

Software systems are becoming increasingly *bloated* to accommodate a wide array of features, platforms and users. This results not only in wastage of memory but also in an increase in their attack surface. Existing works broadly use binary-rewriting techniques to remove unused code, but this results in a binary that is highly customized for a given usage context. If the usage scenario of the binary changes, the binary has to be regenerated. We present **DynaCut**– a framework for Dynamic and Adaptive Code Customization. **DynaCut** provides the user with the capability to customize the application to changing usage scenarios at *runtime* without the need for the source code. **DynaCut** achieves this customization by leveraging two techniques: 1) identifying the code to be removed by using execution traces of the application and 2) by rewriting the process *dynamically.* The first technique uses traces of the *wanted* features and the *unwanted* features of the application and generates their *diffs* to identify the features to be removed. The second technique modifies the process image to add traps and fault-handling code to remove vulnerable but unused code. **DynaCut** can also disable *temporally* unused code – code that is used only during the *initialization* phase of the application. To demonstrate its effectiveness, we built a prototype of **DynaCut** and evaluated it on 9 real-world applications including NGINX, Lighttpd and 7 applications of the SPEC Intspeed benchmark suite. **DynaCut** removes upto 56% of *executed* basic blocks and upto 10% of the application code when used to remove initialization code. The total overhead is in the range of 1.63 seconds for Lighttpd, 4.83 seconds for NGINX and about 39 seconds for perlbench in the SPEC suite.

DynaCut: A Framework for Dynamic Code Customization

Abhijit Mahurkar

(GENERAL AUDIENCE ABSTRACT)

Software systems are becoming increasingly *bloated* to accommodate a wide array of users, features and platforms. This results in the software not only occupying extra space on computing platforms but also in an increase in the ways that the applications can be exploited by hackers. Current works broadly use a variety of techniques to identify and remove this type of vulnerable and unused code. But, these approaches result in a software that has to be modified with the changing usage scenarios of the application. We present DynaCut, a *dynamic* code customization tool that can customize the application at its runtime with a minimal overhead. We use the execution traces of the application to *customize* the application according to user specifications. DynaCut can identify code that is only used in the initial stages of the application execution (initialization code) and remove them. DynaCut can also disable features of the application. To demonstrate its effectiveness, we built a prototype of DynaCut and evaluated it on 9 real-world applications including NGINX, Lighttpd and 7 applications of the SPEC Intspeed benchmark suite. DynaCut removes upto 56% of *executed* basic blocks and upto 10% of the application code when used to remove initialization code. The total overhead is in the range of 1.63 seconds for Lighttpd, 4.83 seconds for NGINX and about 39 seconds for perlbench in the SPEC suite.

Dedication

This is dedicated to my Parents, and my Sister

Acknowledgments

I would like to thank the following people for the all the help, guidance and learning they have provided me during my degree at Virginia Tech:

Dr. Binoy Ravindran for giving me the opportunity to be a part of SSRG, for his constant guidance and support throughout my stay at SSRG. Being a part of SSRG and working with you has been an enriching experience.

Dr. Xiaoguang Wang, for his constant help and advice, patience in explaining concepts and always being there to answer any question. Thank you for introducing me to proper *research* and helping me conclude this work.

Dr. Haining Wang for agreeing to be a part of my committee and providing guidance and assistance in completing this thesis.

I would also like to thank my friends Sidhaarth and Naarayanan for their understanding and support during difficult times.

This work is supported in part by ONR under grants N00014-18-1-2022 and N00014-19-1-2493.

Contents

List of Figures

List of Tables

List of Abbreviations

CRIT	CRiU Image Tool
CRiU	Checkpoint Restore in Userspace
ELF	Executable and Linkable Format
GOT	Global Offset Table
PLT	Procedural Linkage Table
ROP	Return Oriented Programming
VMA	Virtual Memory Area

Chapter 1

Introduction

Existing commodity software systems are designed to support multiple features, platforms, and various users [37]. In actuality, the end-users use only a small subset of these features – leading to software bloat or accumulation of unwanted features not required for the use cases of the end-user. A simple example of software bloat is the increase in the size of the `true` and `false` commands in Unix and Unix-based systems. From 0 bytes and 7 bytes respectively, these commands have gone up in size to 8377 bytes each [20]. This is also true for other commodity software like the Firefox web browser which executes less than 30% of its code [38]. Software bloat not only results in more system memory being consumed but also in an increase in the attack surface of the application. An example of this is the Heartbleed bug [45] in the OpenSSL cryptographic library. Heartbleed resulted from a programming mistake in a popular OpenSSL library making up to 66% of the websites on the Internet that used Apache and NGINX web servers vulnerable. Apart from code bloat, applications tend to use some parts of the code only during startup and never again in the lifetime of the application [16].

1.1 Motivation

Code coverage techniques can be used to find *unreachable* or *dead-code* to identify and remove unused code in the application. Dead code elimination is a compiler optimization that can be performed using static code analysis and data-flow analysis [12]. However, this approach to remove unused code would need to be performed during the software development lifecycle and would be unsuitable for situations where only the application binary is available.

Recent works to remove unused code from the program binary use control flow information [37], [41], [14] to identify the *unused* basic blocks. Some approaches also use user-defined inputs to customize the binaries to retain only the user-desired features in the binary [37], [19], [22]. To remove the *unused* basic blocks, some techniques synthesize a new binary from the constructed control flow graph (CFG) [37] and others replace the code instructions with useless instructions [41]. Temporal Syscall Specialization [16], achieves *dynamic* attack surface reduction by disabling system calls that are used temporally i.e., only in the initialization phase and not in the serving phase of the application. It identifies a *transition point* between the initialization phase and the serving phase of the application. Next, static analysis techniques are used to identify syscalls used only in the initialization phase. These syscalls are eventually disabled using seccomp filters. This technique results in the elimination of dangerous syscalls like `execve` – which can be used to launch dangerous shellcodes.

The overall motivation of our work is that existing works do not target removing unused code at application *runtime* or *dynamically* customize it according to user defined specifications. Existing debloating techniques [37] [19] [14] [41] can remove *unused* code but do not take into account the temporal nature of code execution in an application. Our work also provides a novel approach to disable application features and also re-enable them as and when the user desires.

1.2 Threat Model

Our threat model assumes that the attacker has access to the application binary and can send requests to the server through a remote connection. Our work does not provide any exploit mitigations and only reduces the attack surface of an application. We also assume a trustworthy TCB, OS kernel, and ELF loader. Side-channel attacks and kernel vulnerabilities and mitigations are out of the scope of our work. DynaCut can reduce the viability of code-reuse attacks e.g. ROP attacks [57]. ROP attacks can be defeated with existing OS security techniques like ASLR [57]. But, there exists a more powerful variant of ROP attacks called Return-to-PLT(ret2plt) attacks, which can defeat ASLR. DynaCut can be used to reduce the viability of ret2plt attacks.

1.3 Thesis Contribution

We present DynaCut– Dynamic and Adaptive Code Customization, a framework for *Dynamically* customizing an application at *runtime*. DynaCut can disable features of the application that the user deems unnecessary in a particular usage context of the application. DynaCut can identify and remove application code that is being used only during the startup of the application. DynaCut can also *re-enable* application features that were disabled, providing flexibility to the user. DynaCut does not need the source code of the application to perform these customizations, however, it relies on the execution traces of the application, which can be generated using an off-the-shelf trace generation tool e.g., DynamoRIO [4], Intel PT [40] and Intel PIN [28]. We use the code-coverage tool `drcov` of DynamoRIO [10].

To *dynamically* customize a process, DynaCut uses a *process-rewriting* method that modifies the process address space to add code pages, insert fault handlers into the address space and insert traps in the application binary – providing a rich set of capabilities to the end-user to customize a process.

DYNACUT is built upon the Linux Checkpoint Restore in Userspace (CRiU) project [7]. CRiU provides an infrastructure to *stop* a process, checkpoint its state to disk, and then restore it. We extended the capabilities of CRiU to update the memory contents of the process.

Our contributions are summarized as follows:

- We present the Design and Implementation of DYNACUT;
- We describe how the code-coverage traces generated using DynamoRIO is processed to identify feature-related and initialization basic blocks;
- We built a prototype of DYNACUT and evaluated it on 9 real-world applications; The results show DYNACUT can remove up to 56% of the executed code of server applications.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses the necessary background of our work and also justifies its motivation. Chapter 3 describes the past and related works, Chapters 4 and 5 describe the design and implementation of DYNACUT respectively. Chapter 6 describes the security benefits of DYNACUT. Chapter 7 discusses the performance evaluation of DYNACUT. Finally, Chapter 8 concludes the thesis, discusses the limitations of DYNACUT and identifies future work.

Chapter 2

Background

This chapter aims to provide the necessary background in the context of this thesis.

In Section 2.1, we describe what is software debloating and some existing works in this direction. Next, in Section 2.2, we briefly describe CRiU – upon which DynaCut is built. In Section 2.3, we give a basic overview of signals in linux and describe the `SIGTRAP` signal. In Section 2.4, we briefly review some basic ELF concepts used by DynaCut and finally in Section 2.5, we briefly provide the necessary background of the binary instrumentation framework DynamoRIO [10].

2.1 Software Debloating and Dynamic Customization

To support a wide range of use cases and platforms, applications tend to add on functionalities that may not be required by the end-user of the software. These unnecessary functionalities lead to a waste of memory but more importantly can increase the attack surface of an application [37]. Software Debloating is a technique in which *unused* code or code *bloat* is removed from the application.

Some techniques also allow for the code to be *customized* according to the user specifications. Razor [37] is a debloating framework that customizes the binary based on the users' specifications. Razor relies on user-specified test cases to identify the code necessary/unnecessary for execution. Only the code necessary for the execution will be triggered by the test cases and by tracing the execution, such code can be identified. However, since it is almost impossible to generate test cases to cover all related functionalities, razor also takes into account control flow based heuristics to identify complementary code which cannot be identified using just user-specified test cases. To collect the execution traces, Razor uses both software-based binary instrumentation tools (DynamoRIO [10]) and hardware-based logging tools like Intel PT [40]. Once the traces are collected, Razor disassembles the binary and constructs a partial control flow graph (CFG). To add more related code into the CFG, Razor uses a set of control flow-based heuristics to construct a complete CFG. Once a complete CFG for a set of specified test cases is constructed, Razor then disassembles the original binary and synthesizes the new debloated binary by including the machine code of the necessary instructions. Razor generates a debloated binary customized to the specific needs of the user but is not *dynamically* customizable – if the user-specified test cases change, a new binary has to be generated.

There have also been machine learning based approaches to identify a minimal binary. Chisel [19] and Control-flow trimming [14] are two such examples. Chisel [19] is similar to Razor but uses a Reinforcement Learning-based approach to identify and remove all unwanted functionalities of the application given a high-level specification of the desired functionality. Control-flow trimming [14] generates a contextual CFG by using machine learning based techniques to decide if a branch should be *taken/not-taken*.

Recent research efforts have been aimed at *dynamically* customizing the behavior of the application. Temporal syscall specialization [16], *dynamically* customizes the syscalls that are allowed to be executed by the process. Depending on the execution phase of the application, temporal syscall specialization disables system calls that are no longer required in the current execution phase. They identify a transition point between the *initialization* and the *serving* phase for popular server applications and then disable all the system calls that are not required for the normal execution of the application in its serving phase. They achieve this by using static analysis to extract the application's call graph, identifying all the syscalls used in each phase of the application, and then deploying a seccomp filter to disable all unused syscalls. This results in the blocking of dangerous syscalls like `fork()` and `execve()` which are used only in the initialization phase of the application. These syscalls can be used to launch dangerous shellcodes. By blocking them, the attack surface of the application is reduced.

DamGate [5] is another work that *dynamically* customizes application features. Application features that cannot be completely removed using existing debloating techniques can be blocked by inserting *gates* or *checker* functions in the binary of the application. DamGate performs both a static call graph analysis and a dynamic call graph analysis to generate a complete call graph. Once this complete call graph is generated, execution paths for allowed features are identified and gates are inserted in these execution paths to prevent the execution of unwanted features. If the execution of the function goes beyond the permitted boundary, DamGate terminates the execution of the program. DamGate bears the closest resemblance to DynaCut. But, DynaCut can disable the unwanted code completely and it also allows the user to customize the fault handling when an undesired feature is executed – unlike DamGate which terminates the execution.

2.2 Checkpoint Restore in Userspace

Checkpoint Restore in Userspace (CRiU) is a tool that *checkpoints* an application in userspace. It freezes a running process, saves all its data into a set of images, and when the user wants to restart the process from the point it was checkpointed, it restores the process. CRiU can be used for live container migration, snapshots and remote debugging [7].

CRiU can also be used to reduce the start time of the application. If the application is checkpointed once it finishes starting up, this image can be used to restart the application without waiting for the application to start up again. DYNACUT makes use of this usage scenario – we checkpoint the application once its initialization phase is complete and then since these initialization functions are no longer needed, we can disable them by modifying the CRiU images.

CRiU *dumps* the process by collecting the process and freezing it. If given the PID of the process through the command line, CRiU dumper walks through `/proc/$pid/task/` to collect threads. The children are gathered by reading the `/proc/$pid/task/$tid/children`. The tasks are stopped using the `PTRACE_SEIZE` command while the CRiU dumper collects the process information. Once the process is *frozen*, CRiU collects all the information of the process and writes them to dump files. The VMA areas are parsed from `/proc/$pid/smaps`, the mapped files are read from `/proc/$pid/map_files`, the file descriptors are read via `/proc/$pid/fd` and core parameters of a task like registers, signal handlers etc. are read from the ptrace interface and from parsing the `/proc/$pid/stat` entry. CRiU also injects *parasite* code into the process address space via the ptrace interface. The parasite code collects more information from the process such as the credentials and more importantly, the contents of the memory of the process.

CRiU *restores* the process by morphing itself into the tasks it restores. The process tree is `forked`, the basic task resources are restored and the restorer *blob* restores the memory map of the process – by replacing the memory contents of CRiU with that of the dumped process.

2.3 Signal Handling in Linux

Signals in a Linux process are software interrupts that are sent to the process to indicate some event. Our work makes extensive use of the `SIGTRAP` signal. This signal is issued whenever the program execution hits a `trap` instruction. The trap instruction is a one-byte instruction (`0xCC`) generally used by debuggers to insert a `breakpoint`. For DynaCut, we use the `trap` instruction to customize the process. DynaCut aims to dynamically customize the fault handling behavior of the application. To allow the user to define the fault handling behavior, the default behavior has to be replaced with the customized behavior. We do this by inserting the `trap` instruction at our points of interest and updating the default action of a signal handler. This is achieved by updating its sigaction. A `sigaction` defines how the signal will be handled. `sigactions` can also call specified functions to handle the signal.

```c
struct sigaction {
               void     (*sa_handler)(int);
               void     (*sa_sigaction)(int, siginfo_t *, void *);
               sigset_t   sa_mask;
               int        sa_flags;
               void     (*sa_restorer)(void);
           };
```

Listing 2.1: `sigaction struct definition [24]`

Listing 2.1 shows the `sigaction` structure. DynaCut updates the fault handling behavior by updating the `sa_sigaction` field with the pointer to the signal handler function instead of the default signal handling action. The `sa_restorer` field needs to be replaced with the pointer to the function that makes the `rt_sigreturn`[1] syscall.

[1]The rt_sigreturn syscall returns from the signal handler and cleans up the stack frame [25].

2.4 ELF Basics

ELF (Executable and Linking Format) is a standard file format used for executable files, shared libraries, object code, and core dumps [13]. DYNACUT loads a position independent shared library into a process' address space by loading the executable segments of the ELF file into a newly created virtual address space. DYNACUT accomplishes this by simulating the Linux dynamic loader (`ld.so`) [23]. Next, we describe the execution view of an ELF file – also known as segments, and how they are loaded into the virtual memory for execution.

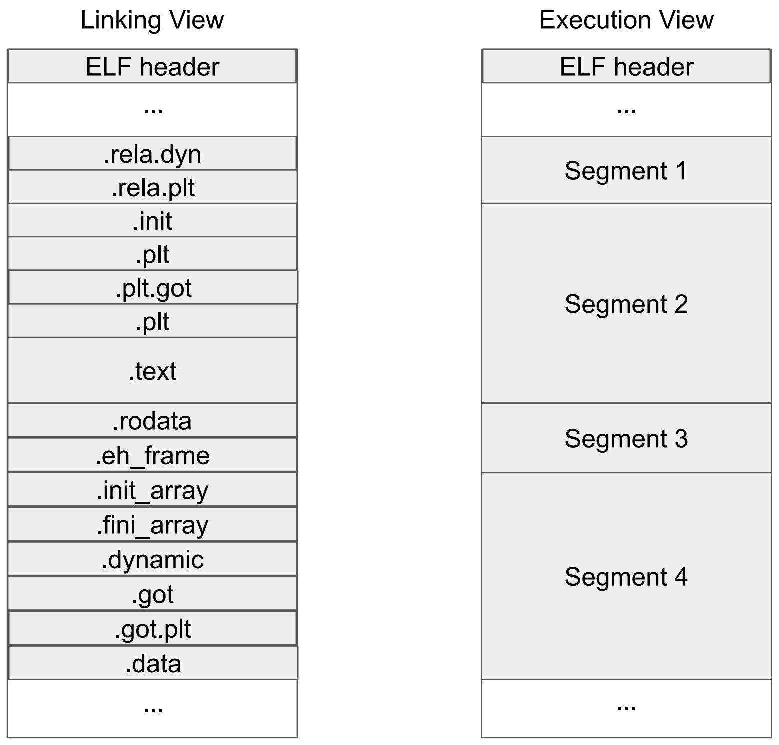

Figure 2.1: Illustration of the Linking View and the Execution View of an ELF file.

2.4.1 Program Loading and Dynamic Linking

ELF files have two kinds of formats depending on whether the ELF is being used for program linking or program execution. ELF `sections` come into picture when an ELF file is used for program linking and ELF `segments` come into picture when the ELF file is being used for program execution. Figure 2.1 illustrates the Linking View and the Execution View of an ELF file.

Various sections in the ELF hold program and control information. Some of these sections are: The `.plt` section holds the procedure linkage table, the `.init` section of the ELF holds the executable instructions that contribute to the process initialization code, the `.got` holds the global offset table, the `.text` section holds the executable instructions of a program, the `.rodata` section holds read-only data that typically contribute to a non-writable segment in the process image. The image represents how the ELF sections are mapped to the ELF segments. In this section, we focus on the execution view of the ELF, i.e., the segment view.

There are nine types of segments in a ELF file. Segments of type `PT_LOAD` specify loadable ELF segments which is described by `p_filesz` and `p_memsz` [48]. These two values describe the size of the `PT_LOAD` segment in-file and in-memory respectively. The value `p_memsz` can be greater than the value `p_filesz` if the `PT_LOAD` segment contains the uninitialized data section (`.bss`). Every one of the `PT_LOAD` segments from the ELF file needs to be loaded into the virtual memory. When the `PT_LOAD` segments are loaded into memory, their relative offsets must be preserved. The difference between any two virtual addresses in memory must match the differences in the ELF file [48]. The difference between *any* virtual address in memory and *any* address in the file is the *base address* of the shared library in the virtual address space. The first `PT_LOAD` segment is loaded at the base address. The second `PT_LOAD` segment is loaded at the end address of the first segment truncated to the nearest multiple of the maximum page size. All the `PT_LOAD` segments are similarly loaded into memory.

2.5 DynamoRIO

DYNACUT uses execution code traces of an application in the first stage to determine which basic blocks need to be removed and also to identify basic blocks that are not required in the current execution phase of the application. DynamoRIO is a framework that provides the capability to *dynamically* analyze and optimize the application at *runtime* [4]. The `drcov` client [9] is a tool that is built upon the DynamoRIO framework. `drcov` collects information about which basic blocks have been executed and writes the results to a per-process logfile. `drcov` is generally used to analyse the code coverage of a particular application. We use the `drcov` tool and the generated execution traces of the application to identify all the basic blocks that have been executed and the order of their execution. DynamoRIO also provides the user the capability to write a custom tool that uses the DynamoRIO framework. We used this capability to develop a tool that prints the first and last execution timestamp of every basic block. We describe how these execution traces were used in section 4.2.

2.6 ROP attacks

Buffer overflow attacks use the buffer overflow vulnerability to overwrite parts of application memory. Buffer overflow attacks work by overwriting a fixed-length buffer with more data than it can handle – leading to overwriting of adjacent areas of memory [53]. Buffer overflow attacks are generally used to execute shellcode leading to arbitrary execution of the application or the attacker gaining root privileges. Some of the defenses against these attacks are: 1. non-executable stack, 2. stack canaries, and 3. Address Space Layout Randomization (ASLR).

Return-to-libc attacks are a type of attack that uses a buffer overflow vulnerability in the application to overwrite the memory with calls to the C standard library `libc`. `Libc` is targeted because it is almost always linked to a program. One example of a return-to-libc attack uses the `system()`[2] libc call. If the address of `system()` and the string "`\bin\sh`" is identified in the process address space, and these addresses are placed in the buffer that is being overflowed, the attacker can open a shell [59]. ROP attacks are a more expressive form of ret2libc attacks and use machine instructions already present in the application memory – called *gadgets*, to enable *arbitrary execution* [59]. Gadgets are chained together to create code sequences that perform arbitrary execution.

Return-to-PLT (ret2plt) attacks are similar to ret2libc attacks where the attacker uses PLT entries of the application to jump into the libc functions. But, ret2plt attacks are more powerful as they can bypass both the non-executable stack defense and ASLR [8].

[2]The system() libc call forks a new child process and executes the shell command specified [26].

Chapter 3

Related Work

This chapter provides an overview of several related works that can have been carried out in the research areas that DynaCut targets. The related works have been split into five categories. First, section 3.1 describes research efforts in the area of software debloating. Section 3.2 discusses some techniques that aim to achieve dynamic attack surface reduction. Section 3.3 discusses approaches to dynamically update software. Finally, sections 3.4 and 3.5 discuss techniques that enforce fault isolation and principle of least privilege.

3.1 Binary Debloating and Customization

There have been several works that introduce different types of software debloating frameworks. The main challenge that is common to all these works is the ability to identify code that needs to be removed. Some of the works that we identified that perform a variation of debloating are Razor [37], Chisel [19], Piece-wise compilation [39], BinTrimmer [41], Feature-based customization [22] and Control-flow trimming [14].

Razor [37] performs debloating *post-deployment*. Razor achieves debloating with the use of binary instrumentation tools and by generating a precise CFG using a heuristic-based approach. Once a precise CFG is constructed, a new binary is synthesized for the user-defined features with only the necessary instructions. **DynaCut** is also a *Post-deployment* tool like Razor and customizes application binaries. However, **DynaCut** can remove *unused* code at application *runtime*.

Another work that attempts to generate a precise CFG is BinTrimmer [41]. BinTrimmer uses a *Iterative CFG refinement* technique to generate a precise CFG – one that identifies all the basic blocks that can be safely removed from the application without affecting its correctness. Once all the basic blocks needed are identified, BinTrimmer removes all the unwanted basic blocks by rewriting them with useless instructions. Even though it does not reduce the application size, it provides a security advantage by reducing the attack surface of the application. **DynaCut** removes unwanted basic blocks by replacing them with useless instructions (`int3`). Although **DynaCut** does not use any CFG creation technique to identify unwanted basic blocks, BinTrimmer's CFG refinement technique can be used orthogonally to **DynaCut**'s technique of identifying unwanted code.

Control-flow trimming [14] is another approach that focuses on generating accurate CFGs for debloating the application. Control-flow trimming generates a *contextual* CFG that uses decision trees to decide whether the branch should be executed or not. The machine learning model is first trained using user-defined execution traces generated by unit testing of the application. To enforce the contextual control-flow integrity, a hash table with key as the context and the value as the yes/no decision is created. Every jump instruction is instrumented to check the hash-table to make the decision to *take/not-take* the branch. **DynaCut** also needs the user-defined test cases to generate application traces to identify *unwanted* basic blocks.

Piece-wise compilation [39] is another approach to include only code that is necessary for the execution of particular user-defined features. **DynaCut** also tries to identify the minimal code that is necessary for the execution of the application – although we do not customize the `libc` binary.

Chisel [19] is a Reinforcement-Learning based approach to software debloating where the wanted features of the application are identified by using a *property test*. Through several iterations, a reduced program is identified. DynaCut also tries to identify the *minimal* code necessary for the execution of the application but without the use of reinforcement learning.

Feature-based customization [22] provides an approach to debloating where unwanted code is input as *seeds* to the debloating framework. The functions corresponding to these seeds are removed by using static dataflow analysis and by removing the call-sites to the identified functions. DynaCut also tries to remove the unwanted code using the user-defined tests cases as input.

Although DynaCut does not perform any CFG construction or optimization, we consider all the above works to be orthogonal to the method of basic block identification used by DynaCut. These methods can be used in conjunction with our simple method – a transition point-based approach to identify unwanted basic blocks, incurring minimal overhead or remove an even larger number of basic blocks – even those that are used at application runtime.

3.2 Dynamic Attack Surface Reduction

In this section, we discuss some related works that aim to achieve attack surface reduction dynamically.

Confine [15] uses static code analysis techniques to analyze a containerized application and identify all the system calls needed by it. Once all the system calls are identified, the remaining syscalls are blocked using seccomp filters. DynaCut tries to identify the minimal set of basic blocks necessary for the application execution and removes all the unnecessary basic blocks – at *runtime.*

BinRec [1] aims to use the execution traces to lift the binary to IR and apply a rich set of transformations to it. Once the transformations have been applied, they can then be lowered back again into binary code. BinRec can be used to reduce the attack surface by using transformations like AddressSanitizer [46] and SafeStack [47]. While **DynaCut** does not lift the binary to IR, similar attack surface reduction can be achieved using **DynaCut**. In the case of BinTrimmer, if the code-usage scenario changes, the binary has to be recompiled, which is not the case with **DynaCut**. **DynaCut** can simply re-enable the code required by the user.

Attack surface reduction has also been demonstrated for OS kernels in *trimming* [27]. Trimming uses kprobes to reduce the attack surface of the kernel. First, analysis kprobes are inserted into the kernel and all the kernel functions that are being used under a policy scope are identified. Next, all the unwanted kernel functions are hooked with enforcement kprobes that kill an application if unwanted access is made to these functions.

DynaCut also achieves a dynamic attack surface reduction – by disabling basic blocks that are not required for the subsequent execution of the application at *runtime.* Hence, **DynaCut** can also be used for dynamic attack surface reduction.

3.3 Dynamic Software Updating

Dynamic Software Updating (DSU) as the name suggests is updating of applications while they are running [56]. There have been several efforts in this direction. We highlight some of them here.

Kitsune [18] is a DSU system for the C language. Kitsune works by inserting update points at program locations that need to be updated. Once the update points are inserted, the update execution phase unwinds the stack of the old threads, loads the entire new program, and resumes execution of the application from the point at which the update took place in the old version.

Ginseng [34] is a system that dynamically patches applications. The applications are specially compiled so that they can be dynamically patched. Then, ginseng compares the old code from the new code, generates the patch, and applies this patch dynamically while the application is running.

Ksplice [21] is a system that allows system administrators to apply patches to the kernel without rebooting. KSplice works by first linking the replacement code into the kernel and then placing a jump instruction in the kernel at the start of the obsolete function – thus directing the execution from the obsolete function to the new function.

Upstare [32] is another work that dynamically updates a running process. UpStare loads the new patch into memory using `dlopen` and also performs datatype updates by unwinding the stack up to the thread-entry point function.

DYNACUT can be used to assist Dynamic Software Update methods by inserting *update points* (`trap` instructions) at points where the application needs to be updated and by replacing them with the new code in the patch.

3.4 Fault Isolation and Domain based Isolation

Fault isolation is a mechanism of isolating untrusted modules to protect their host application [61]. There have been several works in this direction including ARMlock [61], Donky [43], ERIM [52], libmpk [36], Intra-Unikernel Isolation [44] and MonGuard [55].

ERIM [52] is a technique to isolate memory domains that uses the intel MPK extension [33]. With MPK, each virtual memory page of a process can be isolated with one of 16 protection keys – partitioning the process address space into up to 16 isolated domains [52]. ERIM reserves a portion of the address space and makes it accessible only from the trusted part of the application. ERIM ensures isolation by mapping the reserved memory and the general memory of the application to two different MPK domains. The reserved trusted memory is accessible only by the trusted code. Execution flows from the untrusted code to the trusted code and back via *call gates*. The *call gates* enable access to the reserved memory by updating a register and when the trusted code is done executing, the call gates disable access to the reserved memory and transfer control back to the untrusted code. **DynaCut** provides a software-based fault isolation mechanism – instead of *isolating* the fault, **DynaCut** eliminates the potentially vulnerable code entirely.

Other works that use Intel MPK to achieve domain-based isolation are: libmpk [35], Intra-Unikernel isolation [44] and MonGuard [55]. libmpk provides a software abstraction for the MPK hardware extension, MonGuard presents an in-process monitor that is isolated from the rest of the application. Intra-unikernel isolation brings memory isolation inside the *single address space* of the unikernel using Intel MPK.

ARMlock [61], uses the *memory domain* support in ARM processors to create isolated memory domains for a process.

Donky [43] provides a *software-hardware* co-design for in-process isolation that implements *domain* switching entirely in userspace – reducing the switching overhead and the kernel complexity.

3.5 The Principle of Least Privilege

Privilege separation is a technique where the application is divided into several different modules based on the privilege each of them requires [30]. Privilege separation prevents the program from being compromised completely in the event of an attack because each of these modules are executed in their own domains and cannot access parts of the program executing in other domains. Program-mandering (PM) [30] is a technique that implements privilege separation. It achieves this by first constructing a Program Dependence Graph (PDG) by using user-defined specifications. Then the PM performs program analysis to determine the sensitive and insensitive information. The Program is then split by PM into sensitive and insensitive domains. DYNACUT can be used to achieve similar privilege isolation by allowing only certain features of the application to execute in a particular usage context – for example `PUT` and `DELETE` requests to a server can be disabled when the client does not need these requests to be sent to the server.

3.6 Summary of Related Works

We summarize some of the works discussed in the previous sections here. Table 3.1 compares DYNACUT with existing debloating techniques.

Summary of Related Works				
Related Work	No Source Code Required	Consider Process' phase of execution	No binary regeneration required	Disable initialization code
Chisel [19]	✗	✗	✗	✗
Piece-wise [39]	✗	✗	✗	✗
Razor [37]	✓	✗	✗	✗
BinTrimmer [41]	✓	✗	✗	✗
DamGate [5]	✓	✗	✓	✗
Control-flow trimming [14]	✓	✗	✗	✗
Temporal syscall specialization [16]	✓	✓	N.A.	✗
DynaCut	✓	✓	✓	✓

Table 3.1: Comparison of DynaCut with existing debloating works

Chapter 4

Design

As described earlier, the motivation of our work is to *dynamically* customize a process. By customization, we explore how to edit the memory of a process – add or delete certain components of the process' memory. We aim to achieve this with minimal overhead and eliminating as many security vulnerabilities as possible; all while making the tool user-friendly and compatible with a wide range of applications.

This chapter discusses the design and implementation aspects of **DynaCut**. First, we describe the 10000 feet overview of **DynaCut**, next, in Section 4.2, we describe the design of the components that make up **DynaCut**.

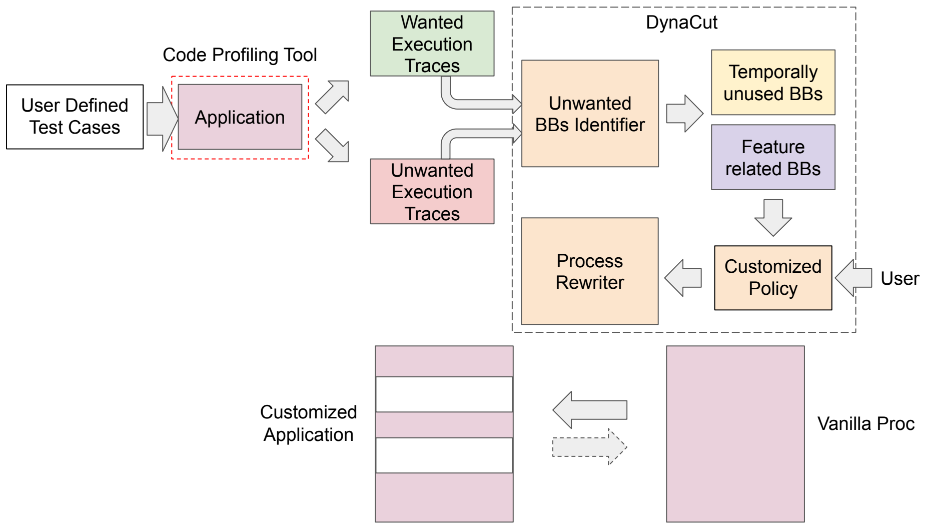

Figure 4.1: Overview of **DynaCut**.

4.1 Overview

To achieve dynamic process customization, we identified two functionalities that DYNACUT must support. First, DYNACUT needs to process the user input – this input describes how the end-user wants to *customize* a process. DYNACUT can remove unwanted features, remove initialization basic blocks , and also add/remove new VMA regions. The second functionality of DYNACUT would be to *edit* the process – perform all the necessary steps that are required to satisfy the user specifications. We provide an overview of these two functionalities of DYNACUT in this Section.

We term the first functionality of DYNACUT as *Undesired basic blocks identification* and the second functionality as *Process Rewriter*. We term the first functionality so because two of the features that DYNACUT supports – feature removal and removal of initialization functions require the identification of basic blocks that are undesired by the user and need to be removed/disabled from the process image. We name the second functionality so because DYNACUT needs to rewrite the process image according to the user specification. We go deeper into these two functionalities of DYNACUT next.

An overview of DYNACUT and its major components : an *undesired code block identifier* and a *process rewriter* is illustrated in Fig. 4.1. The undesired code blocks identifier takes as input the execution traces of the application – generated using the DynamoRIO binary analysis tool [10]. The method to identify undesired basic blocks that DYNACUT uses is very simple – generate a *diff* of the *wanted* and *unwanted* feature logs and identify either the temporally unused basic blocks or the point of transition for the feature to be removed and send it to the next component of DYNACUT– the process rewriter.

The process rewriter is the component of DYNACUT that modifies the memory images of the process to customize it. As described earlier, DYNACUT supports three kinds of process customizations – removal of initialization functions, undesired feature removal, and adding/removing a new VMA region. We describe what the process rewriter does in each of these cases next.

In case of the undesired feature removal, once the point of transition is identified, the process rewriter modifies the process so that once this piece of code is executed, the application either exits using `exit()` or if the user wants it – redirect the execution to another location in the code to handle the exception gracefully.

For removal of initialization functions, DYNACUT identifies all the basic blocks that are used only during the initialization phase of the application and removes them from the process image. By disabling initialization basic blocks, DYNACUT aims to reduce the attack surface available to an attacker to mount code-reuse attacks such as ROP attacks. We describe how DYNACUT can thwart such attacks in the Security Evaluation Section. 6.

The third functionality of the process-rewriter – adding/removing a new VMA region in the process address space and adding any user-specified shared libraries, can be used to handle the exception generated by the above two cases gracefully or to reduce the code even further by unmapping unused VMA regions.

In the rest of this Section, we will describe the design of these two major components in detail and also discuss their low-level implementation details.

4.2 Design

As described earlier, DYNACUT has two major components – the undesired basic blocks identifier and the process rewriter. We describe the design aspects of these two components of DYNACUT in this Section.

4.2.1 Undesired Code Block Identification

DYNACUT's undesired code block identification component mainly uses the execution traces provided by the code coverage client (`drcov`) of DynamoRIO [9]. This component of DYNACUT has to identify the undesired basic blocks for feature removal as well as identify the list of initialization basic blocks. Next, we describe the design of each of these approaches and the differences between them.

Undesired Feature Related Basic Block Identification

To identify the basic block that is responsible for the undesired feature that is to be blocked, DYNACUT uses a simple *diff* based approach. The user first needs to generate a trace with all the desired features and also another trace for all the unwanted features. DYNACUT then compares the two traces and identifies the basic block that is responsible for the execution of the undesired feature as shown in Figure 4.1.

Removal of Initialization Basic Blocks

To identify basic blocks that are used only during the initialization phase of the server application, we need to first identify a *transition point*. The transition point is where the server *transitions* from the initialization phase to the serving phase and the main operations of the server application begin [16].

The *transition point* allows us to determine which blocks are initialization basic blocks – if a basic block has been executed before the transition point, it can be considered an initialization basic block. All other basic blocks executed after the transition point can be considered as essential for the main operations of the server and hence cannot be removed.

The transition point needs to be *manually* identified for every application that is to be tested with DYNACUT. We used a similar approach to identify the transition point as described in [16]. Ghavamnia *et al.* also use a manual approach in identifying the transition point. For server applications like Nginx, the transition point that we considered is similar to the one used by Ghavamnia *et al.* [16] i.e., the transition from the serving phase to the initialization phase takes place once the server's main process forks. We identified the address where the `fork()` code is executed and used it as the transition point for NGINX. Another transition point for NGINX was also identified – the `ngx_worker_process_cycle()` function. For Lighttpd, the `server_main_loop()` function was chosen as the transition point. The basic block at the transition point was identified and all the basic blocks executed before it were identified as initialization basic blocks.

Once the initialization basic blocks are identified, the list of basic blocks to be removed is sent to the process rewriter.

4.2.2 Dynamic Code Customization

DYNACUT can *customize* a process according to the user's specifications. The user can disable an undesired feature, remove initialization code from the application and also update the VMA region of the application either by adding a new VMA region or by deleting existing VMA regions. We elaborate DYNACUT's customization capability in this Section.

Process Rewriting

DYNACUT's process rewriter works on static images created by the Checkpoint Restore in Userspace (CRiU) tool. CRiU freezes a process and stores all the data of the stopped process in a set of image files. DYNACUT processes these image files and modifies them to dynamically customize the process. CRiU was developed to allow for container migration but we only use it to stop a running process and obtain all the process data. Transforming a static process image can prevent potential race–conditions in a dynamic process transformation system [3, 31, 58].

The current capabilities of DynaCut's process-rewriter include: adding/deleting the VMA regions of the process memory, updating the memory contents of the process memory, and adding position independent shared libraries into the process address space. The user can update the memory contents of the process by adding a trap (`int3`) into any basic block that is deemed undesired. Whenever this basic block is executed, the application exits. To handle the `SIGTRAP` gracefully, DynaCut can insert a position independent signal handler library into the process address space. This signal handling library can be customized to handle the exception according to the user's specification. DynaCut's process rewriter can also disable entire basic blocks or unmap an entire page to reduce the attack surface for code reuse attacks. DynaCut can also rewrite the process to update its signal handlers. DynaCut can add position-independent libraries into the process' address space that include a signal handler function and then modify the process CRiU images to handle any exceptions raised by the inserted `int3` instructions. Whenever an undesired basic block is executed, the application execution can be redirected or exited.

Undesired Feature Removal

DynaCut can disable a particular feature if the user deems it unnecessary for the usage context of the application. Once the basic block of the unwated feature is identified in the first phase of DynaCut, a trap is inserted in the first byte of this basic block. A signal handler is also inserted into the process address space in this step. The signal handler is designed to handle `SIGTRAP`. The signal handler can deal with the `SIGTRAP` in several different ways. One action can be to simply call `exit()` to quit the program execution. Users can also program the behavior of the application when there is unintended access to the blocked code. Whenever the blocked code is executed, the fault handler captures the exception and obtains the faulting context. It then updates the instruction pointer by adding an offset value to the fault address so that on signal return, the instruction pointer points to a new location where the application handles the wrong request. For example, when the user wants to disable features in a web server, they can program the fault handler to jump to the code that sends a `403 Forbidden` response.

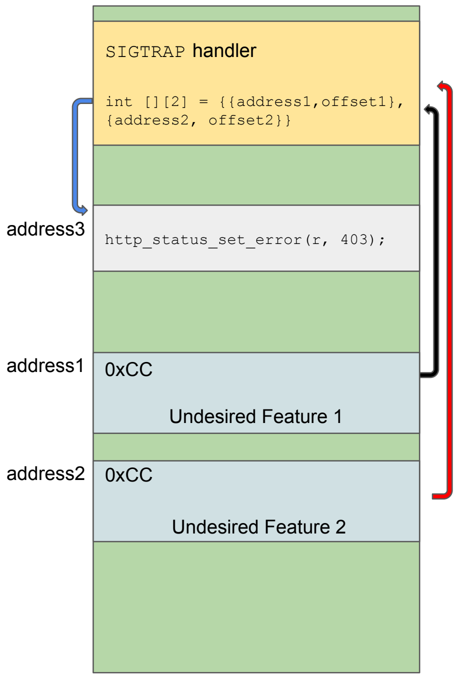

Figure 4.2: Illustration of multiple code features disabling and control flow redirection in DYNACUT.

The feature blocking capability of **DYNACUT** is illustrated in Fig. 4.2. The image shows the layout of the process image that has been modified by **DYNACUT**. As shown in the image, the signal handler library has been added into the process image at a user-defined address. It allows multiple code features to be blocked (by replacing the first byte to an `int3` instruction or `0xCC`). When any of the basic blocks of undesired code blocks are accessed, a `SIGTRAP` is raised, as shown by the red and black arrows. The inserted signal handler then obtains the faulting context and modifies the RIP so that the execution is re-directed to the code block at address3 which makes the server respond with a 403 Forbidden when the blocked feature is accessed, as shown by the blue arrow in 4.2.

To block code features in different locations, the customized SIGTRAP handler updates the instruction pointer based on its original value. To be specific, we store the starting addresses of each unwanted code block – this would be address1 and address2 and the corresponding offsets - this would be (address3 - address1) and (address3 - address2) into the key-value store in the signal handler as shown in Figure 4.2. The offset values are pre-obtained from binary analysis and are embedded into the signal handler through a script.

```c
int trap_map[][2] = {
    #include "removal.h"
};
size_t map_len = sizeof(trap_map) / (2*sizeof(int));

void sig_handler(int sig, siginfo_t *si, void* arg)
{
    ucontext_t *ctx = (ucontext_t *)arg;
    uint64_t rip = ctx->uc_mcontext.gregs[REG_RIP];

    for (int i = 0; i < map_len; i++) {
        if ((rip & MASK) == (trap_map[i][0] + 1)) {
            rip += (trap_map[i][1] - 1);
        }
    }
    ctx->uc_mcontext.gregs[REG_RIP] = rip;
}
```

Listing 4.1: Example of a SIGTRAP handler for different trap locations.

An example of the signal handler code that can be used to achieve the feature removal is listed in Listing 4.1. The listing also shows how the faulting context is obtained (using `ucontext`) and how the `RIP` is updated.

Removal of Initialization Functions

For most long-running server applications, the initialization code will not be used after a certain point in the program execution. Using the execution log with timestamps cannot provide us an accurate location in the code of this transition. To solve this issue, we extended DYNACUT's ability to insert traps and add signal handlers into the process image to dynamically locate the transition point.

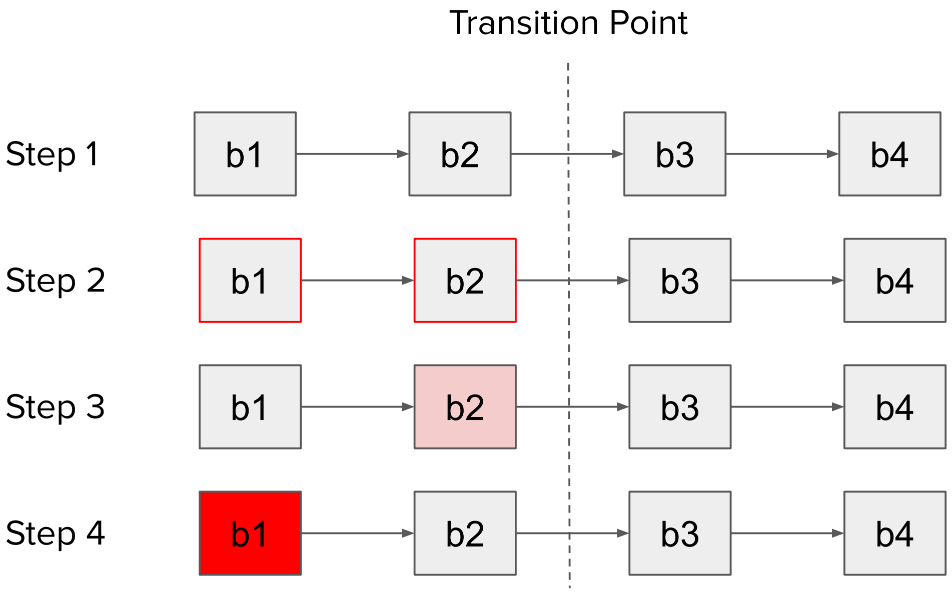

Figure 4.3: Illustration of steps in removal of initialization code

To discover initialization basic blocks, we first use the execution log to find an approximate transition point that never appears in the main execution loop of the server application (step 1 in figure 4.3). Next, we add an `int3` instruction at the beginning of each of the basic blocks executed before this chosen initialization point (step 2 in figure 4.3). We then restore the process. If the basic block with the `int3` instruction is executed, the execution will be trapped by the customized signal handler. The signal handler saves the executed basic block locations to a configuration file to that can be used to finalize the initialization code blocks to be removed – illustrated in step 3 of figure 4.3. Finally, all the identified basic blocks which did not execute in step 3 are removed (step 4 in figure 4.3).

Initially, we tried to find an exact list of initialization basic blocks using only the execution traces. However, there were a few exceptional cases of basic blocks that are never executed after the initialization phase in the execution traces, but are executed when we restore the process (e.g., `ngx_signal_handler()` in Nginx). We suspect that these code blocks are invoked by the CRIU restoration process. Using the dynamic initialization code discovery mechanism described above, any similar corner cases can be avoided.

Chapter 5

Implementation

In this chapter, we discuss how DynaCut was implemented.

DynaCut builds upon CRIU [7] and DynamoRIO [10]. Checkpoint Restore in Userspace (CRiU) is a project that provides the capability to checkpoint/restore in userspace and forms the backbone of DynaCut. CRIU provides DynaCut the capability to stop a running process and save its memory pages, registers, opened files and network connections to a set of process images. This set of images are modified in the later stages of DynaCut. It can also be used to restore the process exactly as it was at the time of the freeze. CRIU is especially useful for transforming stateful programs with live connections – which includes most web servers and key-value stores, as it provides `TCP_REPAIR` support which can re-establish the saved TCP connection.

Once the server starts, it can be checkpointed at a point where the user can assume that the initialization phase is complete. Once the server starts accepting requests, it can safely be assumed that the initialization phase of the server is complete and the server can be checkpointed. At this point, DynaCut can be used to remove initialization code –producing a customized process image of the server without any initialization code (Section 4.2.2).

The checkpointing, customization, and restoration is a one-shot process that can be performed by running a script provided by DynaCut. If the end-user does not want the modified process anymore, the vanilla images created when checkpointing the process can be used to restart the process in the state it was when the checkpointing was applied.

Subsequent parts of this Chapter describe how this was implemented. First, Section 5.1 describes the modifications made to CRiU to implement DynaCut. Next, Section 5.2 describes the modifications made to set of image files to add a signal handler into the process address space. Section 5.3 describes how execution traces are generated and how the `SIGTRAP` signal can be handled. Finally, Section 5.4 provides an overall picture of the implementation.

5.1 Modifications to CRIU

To implement DynaCut, several changes were made to CRIU. For example, CRIU only dumps the pages of the process that are marked anonymous to save bandwidth when transmitting these image files for process migration. Pages that are marked `ANONYMOUS` are not backed by a file. One of the uses of these kinds of pages is to use it as a shared memory region. Pages that are marked `PRIVATE` are file-backed pages but any modifications to the in-memory copy are **not** written back to the file. The code pages of a process are usually marked `FILE PRIVATE`, which means that these pages are file backed in-memory copies and any changes to the in-memory copy are not written back to the file. Code pages need not be saved by CRiU because file-backed memory can be reconstructed by the page fault handler when a restored process tries to access the virtual memory again. In DynaCut's implementation, we added an option in the `criu/mem.c` file to dump the private and executable pages marked `PROT_EXEC` and `FILE_PRIVATE`. This was done because we add traps in the executable section of the application – to customize these regions of memory, we need to modify their in-memory copies.

Extensions were also made to the CRIU image tool (CRIT) to support process rewriting. CRIT provides a user-friendly interface to examine the process images which are encoded in the protocol buffer format [17] – by providing APIs to decode these images to human-readable JSON files. CRIT can also read the saved images and: print all VMA memory regions of the application(`x` and `mems`), print all the *dumped* memory regions (`x` and `rss`) and encode the JSON files back to the protobuf format (`encode`).

Extensive changes were made to the current capabilities of CRIT to provide user-friendly APIs to customize the process – some of these APIs add support to: update memory contents, add new regions, unmap existing VMAs and insert position-independent shared libraries into the virtual address space. These features can be used by the end user to customize the code. As mentioned in Section 4.2.2, these extended features support inserting the signal handler into the process address space, configuring the images for removing initialization basic blocks, removing initialization basic blocks, printing shared library information etc. Our extension to CRIT also includes support to remove a single basic block/function if given its VMA base address, its size, and its file offset.

We also add options to CRIT to configure the images for feature removal and removal of initialization basic blocks. To configure the images for feature removal, DynaCut creates a `removal.h` file that contains the file offsets of the features to be removed and the *relative offsets* of these locations in the code where the execution should jump, as described in Section 4.2.2. For removing initialization basic blocks, the configure option creates a `removal.h` file that contains the addresses of the locations where the trap is added and the data bytes at each of these locations that are replaced with `int3`. The signal handler is compiled using these `.h` files and the data in these `.h` files is loaded into a key-value pair list for run-time reference (code listing 4.1).

5.2 Adding Signal Handler into the Process Image

For both the functionalities of DynaCut, a signal handler needs to be added into the process address space that can handle `SIGTRAP`. Our extension to CRIT accomplishes this by modifying the CRIU images and adding the signal handler which is compiled as a shared library, into the process address space. DynaCut decodes the vanilla images using existing CRIT APIs, transforms the process images as required and encodes them back into the protobuf format, again using the existing CRIT APIs. When decoded, all the CRiU images can be referenced as a dictionary of lists in Python [11]. Any modifications to the images would involve adding a new entry in the dictionary of lists. In particular, DynaCut rewrites the following images:

The `core` image file: This file contains saved process information including signal handlers for the process, signal masks, register values at the point of freeze etc. DynaCut modifies the `core.img` file and adds the signal handler address, the restorer address and the signal mask into the `SIGTRAP` sigaction entry in this file. For the signal restorer address, we add the restorer code in the code pages of the signal handler library and add its address to the sigaction entry. The restorer code is a 9 byte code that makes a `rt_sigreturn` syscall [1]. The signal handler address entered into the sigaction entry is calculated by adding the file offset of the signal handling function with the VMA base address given by the user.

The `pages` and `pagemap` image files: The raw data contained in the pages of the process address space are stored in the `pages.img` binary file. The `pagemap.img` file contains information about the virtual memory regions populated with the raw data. Each of the entries in the `pagemap.img` file contains the address of the VMA, the number of pages populated with data and the VMA flags. The `pagemap.img` is used by CRiU to index the raw `pages.img`

[1]The rt_sigreturn syscall returns from the signal handler and cleans up the stack frame [25]

content. To load a shared library into the target address space, new VMA entries need to be created in the `pagemap.img` file and new memory pages containing the library code and data need to be inserted in the `pages.img` file. We use an open-source library pyelftools [51] to parse the shared library ELF. DYNACUT's process rewriter parses the shared library ELF, locates segments marked `PT_LOAD`, calculates the size of each of the `PT_LOAD` segments, and obtains the offset at which each of these need to be loaded. The process rewriter also reads the raw data from each of the `PT_LOAD` segments in the shared library ELF using pyelftools. This data is then added into the CRIU `pages.img` file by creating new raw-data pages and ordering them according to the `pagemap.img` file. Then, DYNACUT loads the ELF at any user-specified 64-bit user-space address that is not already used by the process. Upon restore, this new region of memory will contain the shared library code.

The `mm` image file: This file provides information about the virtual memory regions of the application. The information includes the start address, end address, file offset, shared memory id, permission flags, and status flags of the VMA regions of the application. This file differs from the `pagemap` image file in the manner that the `pagemap` image file only contains details about pages that are populated with data; whereas the `mm` image file contains information about all the VMA regions of the application. DYNACUT modifies the `mm` image file and adds the VMA information (start address, end address, file offset and the corresponding permissions) for the shared library added into the process address space.

DYNACUT also performs global data relocations and procedure linkage table (PLT) relocations [29] with respect to the user-specified VMA address for the library to resolve the addresses of symbols used by it. Global data relocations are performed by adding the VMA base address of the library to the `st_value` field of the symbol. As illustrated in Figure 5.1, for PLT relocations, DYNACUT first finds the external `libc` function symbol offsets from the `libc` binary. Next, the runtime VMA base address of `libc` is added to these symbol offsets and

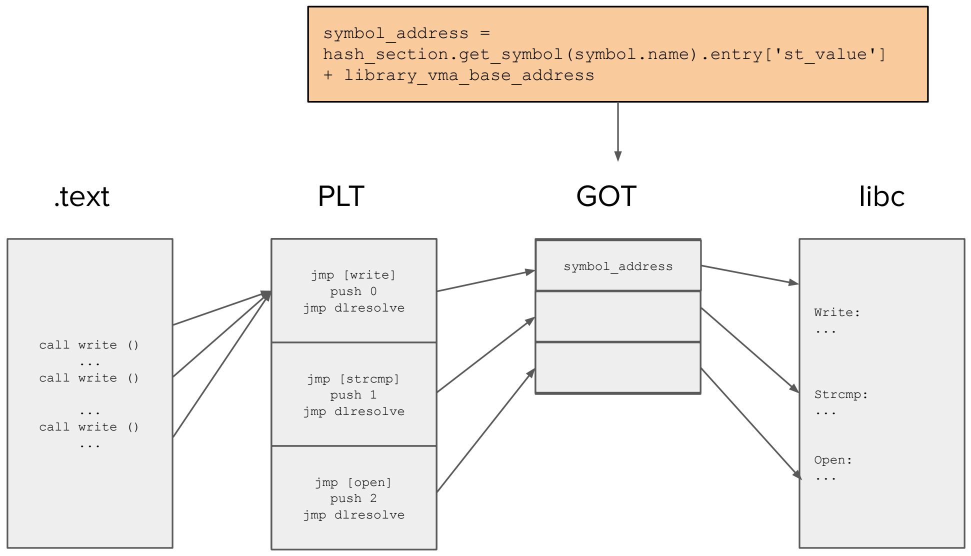

Figure 5.1: Illustration of resolving `libc` symbol addresses

then, these addresses are written to the global offsets table (GOT) [29] of the signal handler library. It is to be noted that these relocations only need to be performed for the global data and `libc` functions used by the signal handler in the shared library.

5.3 Trace collection and `SIGTRAP` handling

To generate the traces used by DYNACUT, the user should pre-run all the test cases required for the normal execution of the application with the DynamoRIO `drcov` client. `drcov` then records all the executed basic blocks and their addresses. DYNACUT also requires the end-user to generate as many use cases as possible for both wanted and unwanted features and generate traces for both of them. Existing fuzzing techniques can also be used to achieve higher code coverage and ensure that all the required basic blocks for a particular use case have been executed. [60]. We also developed a custom DynamoRIO tool to print the execution trace (basic blocks) with timestamps. The traces generated using this tool can be used for identifying the initialization code and by extension, the transition point between the initialization and the serving phase.

After all the feature-related and initialization code blocks have been identified, DynaCut writes the file offsets for each of the identified basic blocks into a configuration file (i.e., `removal.h` in code listing 4.1). End-users need to compile the fault handler using this file and the fault handler skeleton code. The contents of the `removal.h` file are added as a global array in the shared library ELF when the signal handler is compiled. This generates a shared library for the DynaCut process rewriter to load. End-users can also specify a policy for the cases in which the undesired code blocks are accessed. Server applications typically have an event-loop that dispatches the request to the corresponding handler, and usually there is a code that is executed to send a negative response to a request. By identifying such a location manually, the end-user can redirect the blocked code accesses to the handler that sends a negative response. For example, we redirected the control flow to the wrong request handler that sends a `403 forbidden` response for the Lighttpd web server. In the case of NGINX, we redirected control to a location that sends a `405 not allowed` when a unwanted request is received. These locations can be customized by the end user and can vary for different applications.

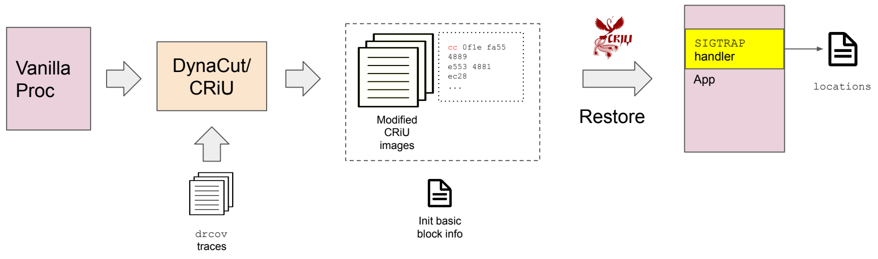

Figure 5.2: Illustration of configure init step of DynaCut.

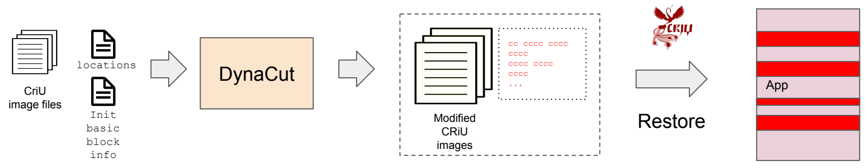

Figure 5.3: Illustration of remove init step of DynaCut.

5.4 Putting them together

This section summarizes the steps that the end-user needs to take to customize the process.

DynaCut's removal of initialization basic blocks capability is summarized in Figures 5.2 and 5.3. In figure 5.2, DynaCut uses the `drcov` traces to modify the CRiU images by adding a trap in the first byte of all the identified *initialization basic blocks*. This modified process is then restored to find all locations of code being used after CRiU restore.

The second step is illustrated in Figure 5.3. DynaCut uses the locations file, and a list of all the initialization basic blocks to *permanently* remove the initialization basic blocks. This modified process can then be restored with the *initialization basic blocks* removed. Most of these steps can be automated using scripts from DynaCut. DynaCut can also re-enable the removed features. All the end-user needs to do is save the removed code blocks (and their addresses) to external storage. The process snapshot enables DynaCut to support multi-process applications and multi-threaded applications without having to deal with the race conditions. To avoid the cost of saving the process image to a hard disk, we save the process images into an in-memory filesystem – tmpfs [42].

Chapter 6

Security Evaluation

In this chapter, we discuss the security benefits that DynaCut can provide to server applications from code-reuse attacks.

Section 6.1 discusses how DynaCut reduces the attack surface of the application. Section 6.2 discusses a variation of the ROP attack that makes NGINX vulnerable and we also discuss how DynaCut can be used to thwart it.

6.1 Reducing the Viability of Code Reuse Attacks

In this section, we discuss how DynaCut can reduce the viability of code-reuse attacks like ROP attacks. DynaCut can remove unused basic blocks at *runtime* – by identifying a transition point between the serving and the initialization phase. This results in the reduction of the code in the application that can be used for code-reuse attacks. By disabling basic blocks that are no longer needed by the application, we reduce the viability of code-reuse attacks.

DynaCut can also remove PLT entries of the application. Since DynaCut can remove most initialization code in a checkpointed application, we also evaluated whether it can remove PLT entries in the application. DynaCut can disable up to 43 out of 56 **executed** PLT entries in NGINX – which are no longer needed after the initialization phase of NGINX is executed. If the transition point `ngx_worker_process_cycle`, is used for NGINX, it also disables the `fork` basic block as the worker process has already been created. Even the PLT entry to `fork` was disabled – thwarting any attacks that could use the `fork` libc call.

A similar evaluation was carried out for Lighttpd and we removed about 33 of the 57 *executed* PLT entries. Some of the PLT entries that were disabled in Lighttpd are the PLT entries for `strcmp`, `dlopen` and `socket`.

The disabling of PLT entries sets DynaCut apart from other works in this direction. While other works can remove *unused* basic blocks and by extension, unused PLT entries, DynaCut can remove *executed* PLT entries that have been used only in the initialization phase of the application.

6.2 Blind ROP (BROP) and NGINX

Blind ROP (BROP) is a variation of the ROP attack that *remotely* locates ROP gadgets in an application [50]. It is an attack that can be used to even hack binaries that are not in possession of the attacker. The two requirements for a BROP attack are a stack vulnerability and a server application that restarts worker processes after a crash – like NGINX. NGINX presents a stack-based buffer overflow vulnerability (`CVE-2013-2028`). This CVE allowed remote attackers to either cause a Denial-of-Service (DoS) by crashing the server or execute arbitrary code using a chunked Transfer-Encoding request with a large chunk size, triggering an integer signedness error and a stack-based buffer overflow [54] [6]. Thus satisfying both the requirements to mount the attack.

To mount a BROP attack, the attacker first needs to find *gadgets* to perform a `write` syscall – after which the attacker can transfer the binary via the network and find even more gadgets to mount other ROP attacks. The BROP attack uses the `write` PLT entry and a entry for a `libc` function that uses the `rdx` register to set the length of the `write`. The authors of the BROP attack use `strcmp()` as the function that uses the `rdx` register. In summary, to mount a BROP like attack, the application binary needs to be scanned for `gadgets` and also the required PLT entries need to be available.

DYNACUT thwarts such attacks in two ways – 1. It removes about 56% of the executed basic blocks in NGINX, reducing the amount of code available for code-reuse attacks and hence making it difficult to find ROP gadgets to mount an attack. Second, since NGINX also disables about 76% of the *executed* PLT entries for NGINX, it becomes difficult for the attacker to find PLT entries to mount the attack.

If the attacker circumvents both these defenses, DYNACUT makes it difficult to mount a BROP attack on NGINX by disabling the `fork()` basic block. The first attempt to crash a worker process will result in the server crashing, thus preventing its exploitation.

6.3 Number and Size of Basic Blocks Removed

As one of the use cases of DYNACUT is to reduce the code available for code-reuse attacks in the application by removing unused basic blocks, we measure the number of basic blocks removed by DYNACUT and the amount of code removed by DYNACUT in this section.

6.3.1 Number of Basic Blocks Removed

We compare the number of *executed* basic blocks removed by DYNACUT for each of the test applications in this section. Figure 6.1 illustrates the basic blocks removed with three different metrics – 1. The number of basic blocks executed for a given user input, 2. The

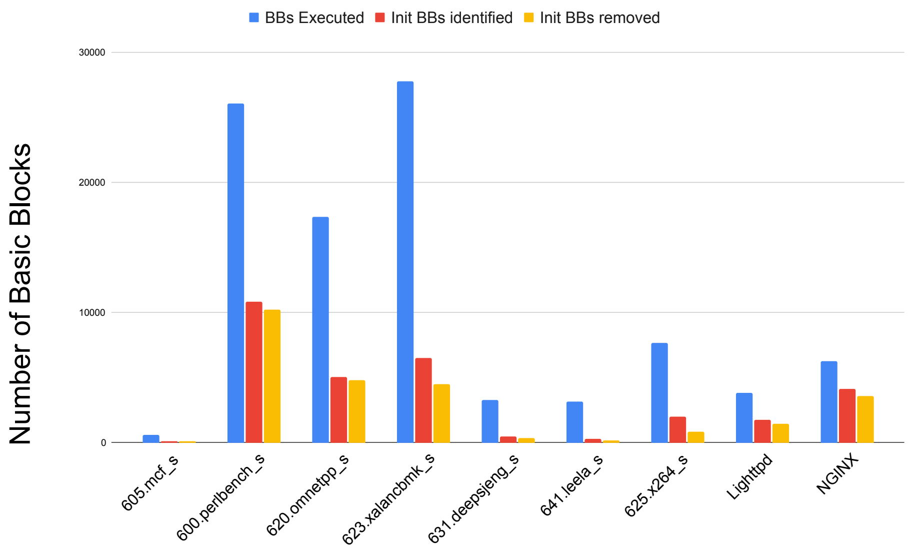

Figure 6.1: Comparison of Basic blocks removed by DynaCut

number of initialization basic blocks identified and 3. The number of basic blocks finally removed. The number of initialization basic blocks identified differs from the number of initialization basic blocks removed because some basic blocks may be present in the whitelist created in the configuration step of initialization basic blocks removal or they could be overlapping basic blocks – in which case we do not remove any of the two overlapping basic blocks.

As illustrated in the graph, for NGINX, DynaCut removes upto 56% of the *executed* basic blocks and about 46% of the executed basic blocks for Lighttpd. In the case of the intspeed benchmarks, we remove the highest percentage of the basic blocks for `perlbench` – with about 41.4% of the executed basic blocks removed.

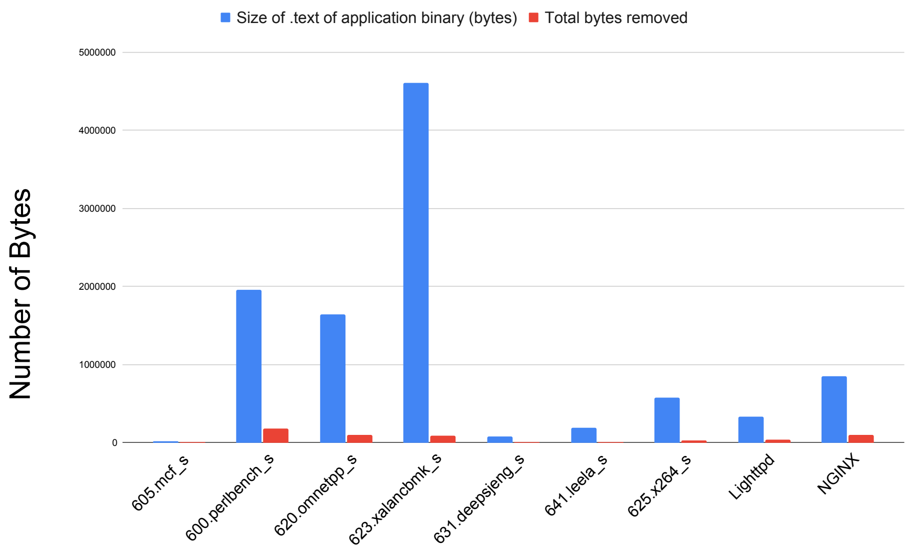

Figure 6.2: Number of bytes removed vs .text size of the application

6.3.2 Number of Bytes Removed

Apart from the number of basic blocks removed, we also measured the amount of code removed by DynaCut relative to the `.text` section of the application. This is necessary because while the number of basic blocks removed measures the basic blocks removed w.r.t. the number of `executed` basic blocks, measuring the amount of code removed gives us a measure of how much code is actually removed by DynaCut.

As illustrated in Figure 6.2, for NGINX, we remove about 11% of the `.text` section of the application and for Lighttpd, we remove about 9% of the `.text` section of the application.

Chapter 7

Performance Evaluation

In this chapter we describe the evaluation of DynaCut. Particularly, we answer the following three questions:

- What is the performance overhead DynaCut incurs to modify images for the feature removal customization?
- What is the performance overhead DynaCut incurs to modify images for the initialization functions removal customization?
- How effective is DynaCut in removing initialization basic blocks – more specifically, what is the number of basic blocks removed and their size in the `.text` section of the application?

This chapter is organized as follows: Section 7.1 describes the experimental setup we used to evaluate DynaCut. Section 7.2 describes the overhead incurred by DynaCut to modify the images for different features of DynaCut. Section 7.3 provides a Summary of our evaluation.

7.1 Experimental Setup

All of our evaluation was performed on a 64-bit system with an Intel i5-10210U CPU clocked at 1.60GHz and 16GB of RAM. The system runs Ubuntu 20.04 with the Linux kernel version at 5.8.0. We evaluated DYNACUT with two server applications, an in-memory key-value store (Redis), and 7 out of 10 SPEC2017 intspeed benchmarks. The server applications we use are Lighttpd and NGINX. Lighttpd is a lightweight web server, has an event-driven architecture, and is a single process application. NGINX is a high-performance web server whose architecture allows for a single master process and multiple worker processes. For the evaluation of DYNACUT, we configure NGINX to use a master process and a single worker process. The SPEC2017 intspeed suite of benchmarks was chosen to evaluate DYNACUT with CPU/Memory-intensive applications and `C++` applications. Redis was used for the evaluation of the feature-removal capability of DYNACUT.

To evaluate DYNACUT on web server applications, we set up the WebDAV (Web Distributed Authoring and Versioning) extension for both the server applications. We sent the `GET`, `PUT` and `DELETE` requests to the server applications using `curl` commands. The `GET` request is used to retrieve resource representation/information from the server, the `PUT` request is used to update an existing resource and the `DELETE` request, as the name suggests, is used to delete a resource [49]. We considered the `PUT` and `DELETE` commands as ”undesirable” for the feature blocking capability of DYNACUT for the server applications. For Redis, we considered the `set` command as undesirable. Since the SPEC 2017 suite of benchmarks does not have a command-line interface to ”features”, we evaluated them only for the initialization functions removal capability of DYNACUT.

DYNACUT relies on the end-user to generate the execution traces for both feature blocking and initialization functions removal. Therefore, the user should execute all the test cases for the desired functionality when generating the traces. For our evaluation, we executed the server application with `GET`, `PUT` and `DELETE` requests and also executed 100,000 requests with `apachebench` [2]. This was done to ensure that the traces generated covered the basic functionality of these applications completely. For Redis, we used `redis-benchmark` to send 1000 `set` and `get` requests to generate the traces. For the SPEC `intspeed` suite of benchmarks, the traces were generated by executing the benchmarks until completion. All the traces were generated using the `drcov` tool of DynamoRIO.

7.2 Overhead Analysis

The overhead analysis of DynaCut involves measuring the time DynaCut takes to modify images according to the user specification. We evaluated DynaCut for two types of overheads: 1. The overhead to modify images for the feature removal customization and 2. The overhead incurred to modify the images for the removal of initialization functions customization.

7.2.1 Overhead for Feature Removal Customization

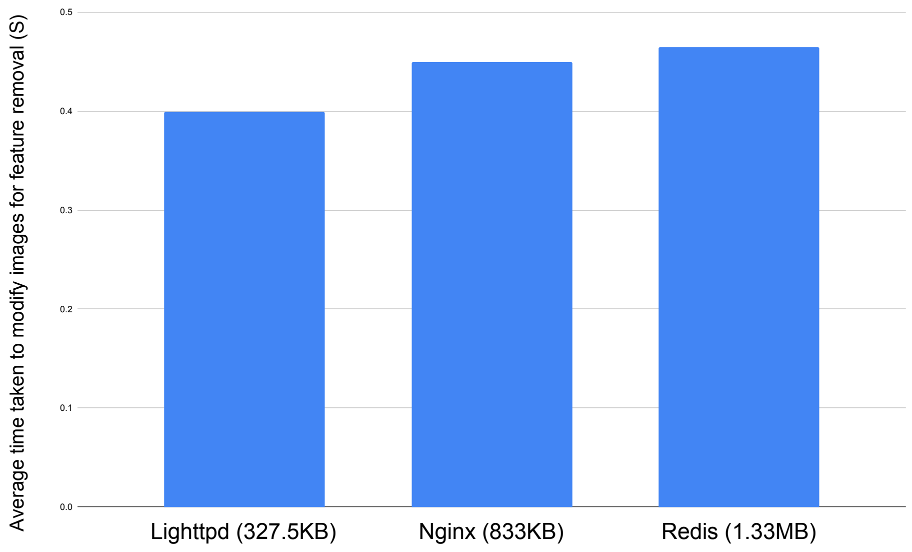

Figure 7.1: Overhead of modifying images for feature removal

For the feature removal optimization, as described in Section 4.2, we identify the basic block for a particular feature and then insert a trap to disable it. This section discusses the overhead incurred to modify the images to insert the trap in the executable and add the signal handler into the process address space. We perform this evaluation for the server applications – Lighttpd and NGINX and the in-memory key-value store Redis.

As shown in Figure 7.1, the overhead incurred for Lighttpd is about 0.4 seconds with a standard deviation of 0.012 seconds and for NGINX, it is about 0.45 seconds with a standard deviation of 0.028 seconds. For Redis, the overhead incurred for the same customization is about 0.465 seconds with a standard deviation of 0.004 seconds. The size of the `.text` section of these applications is also shown in the graph. The overhead also includes the time taken to compile the `removal.h` file created by DynaCut with the signal handler and also the time taken by CRiU to dump these applications.

7.2.2 Overhead of Removal of Initialization Basic Blocks Customization

The Feature-removal customization has two parts: 1. We configure the initialization functions removal, which gives us the final list of initialization basic blocks to be removed, and 2. The actual removal of the identified basic blocks. We discuss both of these overheads in the subsequent subsections.

Overhead of Configuring the Images for Removal of Initialization Basic Blocks

The first part of initialization code removal involves adding a signal handler into the process address space and adding traps into the first byte of all identified basic blocks. Figure 7.2 shows the time taken by DynaCut to modify the images for 8 different applications. The size of the `.text` section of the application is also included in the plots. As shown in the plot – the time taken to modify the images for Lighttpd and NGINX is 0.747 seconds

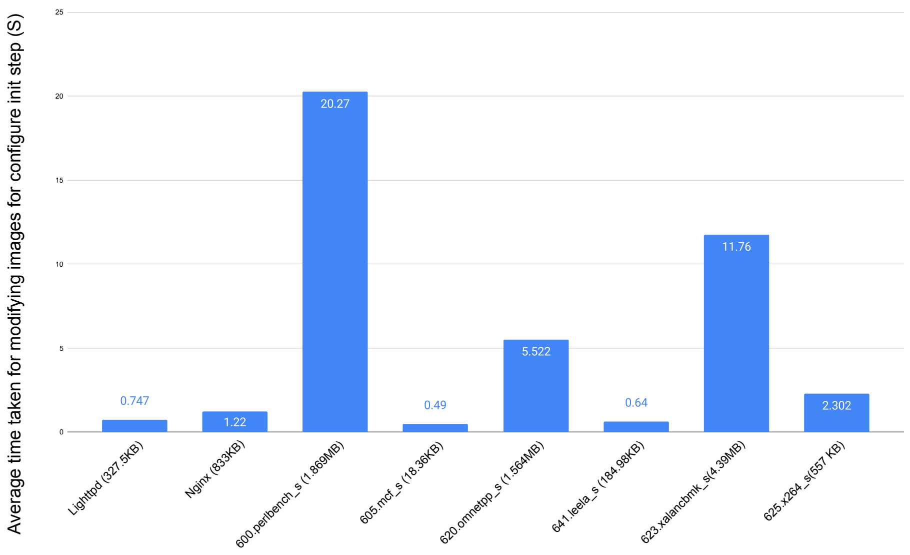

Figure 7.2: Overhead to configure images for init removal

and 1.22 seconds respectively. The evaluation of the SPEC suite of benchmarks differed slightly from the evaluation of the server applications. Since the applications in the SPEC suite of benchmarks do not have a *initialization* point, we chose an arbitrary initialization point for these applications. The 605.mcf_s and 641.leela_s SPEC applications were small when compared to other SPEC applications – leading to low overhead. The overhead to modify the images depends on various different factors – the initialization point chosen, the length of the `drcov` trace of the applications, and the size of the CRiU dumped images. This is illustrated in the plots of `600.perlbench_s` and `623.xalancbmk_s`. Even though both the applications have a similar `.text` section size and a similar CRiU dump folder size (184MB vs 191MB), the time taken to modify `600.perlbench_s` is about 9 seconds more than the time taken to modify `623.xalancbmk_s`. This is because, we chose an initialization point that is much deeper for `600.perlbench_s` than for `623.xalancbmk_s`, causing the extra overhead. Choosing an initialization point at a similar depth for both the applications reduces the time taken for `600.perlbench_s` to 16.7 seconds. This brings to light the fact that the end-user can choose a low initialization point to reduce the overhead incurred, but would have to make sure that all the targeted initialization basic blocks are removed. The 631.deepsjeng_s benchmark was *not* evaluated for the overheads of initialization basic blocks removal because the all the image dumps were modified on a tempfs folder and the CRiU image dump of 631.deepsjeng_s is about 6GB. When DYNACUT was run on this image, it resulted in out of memory error on our system.

The standard deviation of each of these measurements are as follows: 1. Lighttpd – 0.0268 seconds, 2. Nginx – 0.020 seconds, 3. 600.perlbench_s – 1.02 seconds, 4. 605.mcf_s – 0.0087 seconds, 5. 620.omnetpp_s – 0.003 seconds, 6. 623.xalancbmk_s – 0.022 seconds, 7. 625.x264_s – 0.05 seconds, 8. 641.leela_s – 0.0188 seconds.

Overhead of Removing Initialization Basic Blocks

Once the final list of initialization basic blocks is identified, DYNACUT modifies the CRiU images by replacing the basic blocks with int3 instructions. We describe the overhead incurred by DYNACUT to remove the initialization basic blocks in this section.

As shown in Figure 7.3, the overhead to remove the basic blocks for Lighttpd and NGINX is about 0.89 seconds and 3.6183 seconds respectively. The overhead incurred in this case would depend on the number of basic blocks that need to be removed, which implies that it would indirectly depend on the initialization point chosen. For 600.perlbench_s, we identified about 10808 basic that can be removed and for 623.xalancbmk_s, we identified about 6497 basic blocks that can be removed. The overhead incurred is directly proportional to the number of basic blocks that have been identified, as is evident in the graphs of 600.perlbench_s and 623.xalancbmk_s. DYNACUT takes about 4 seconds more for perlbench than xalancbmk to remove the initialization basic blocks – which can be attributed to the larger number of

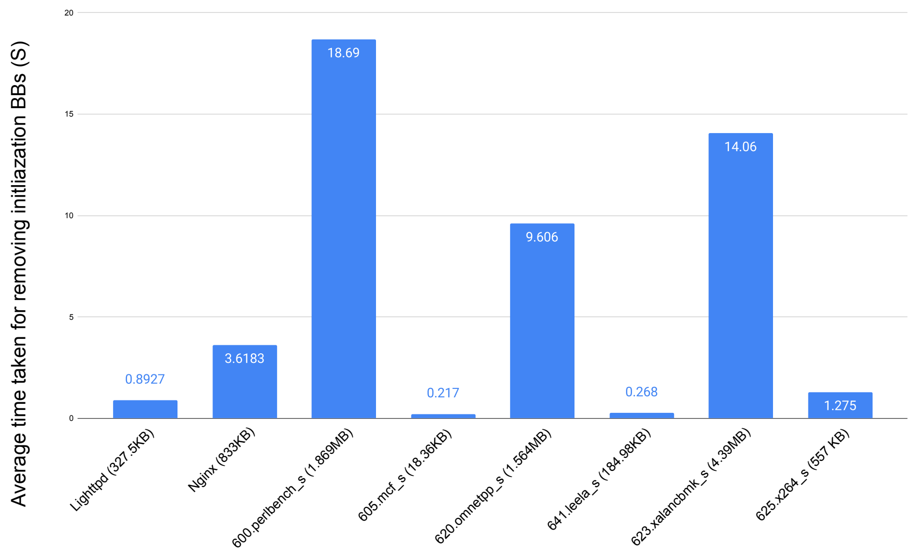

Figure 7.3: Overhead to remove initialization basic blocks

basic blocks identified in the case of `perlbench`.

Again, `631.deepsjeng_s` was not evaluated on this parameter due to the out of memory error.

The standard deviation of each of these measurements are as follows: 1. Lighttpd – 0.0065 seconds, 2. Nginx – 0.112 seconds, 3. `600.perlbench_s` – 1.559 seconds, 4. `605.mcf_s` – 0.00839 seconds, 5. `620.omnetpp_s` – 0.0155 seconds, 6. `623.xalancbmk_s` – 0.370 seconds, 7. `625.x264_s` – 0.0387 seconds, 8. `641.leela_s` – 0.00152 seconds.

Total Overhead

The total overhead to modify the images for removing initialization basic blocks is described in this section.

As illustrated in figure 7.4, the total overhead to modify Lighttpd is about 1.63 seconds and the overhead incurred to modify NGINX is about 4.83 seconds. DynaCut incurs the highest overhead for the `600.perlbench_s` images, which take about 39 seconds to modify.

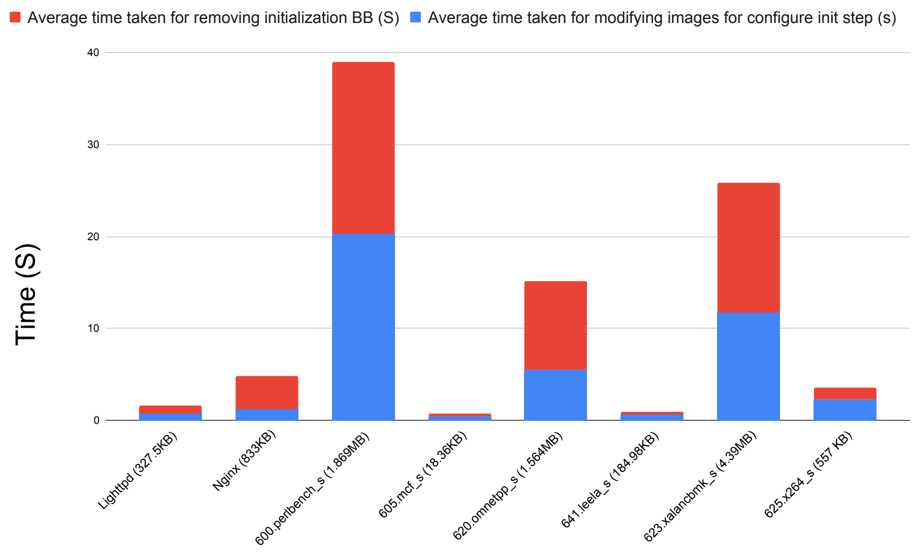

Figure 7.4: Total overhead to remove initialization basic blocks

7.3 Evaluation Summary

In summary, **DYNACUT** can provide a low-overhead method to *dynamically* reduce the code available to mount code-reuse attacks for an application. Other attack surface reduction methods – [37], [16] provide effective methods but incur a relatively high overhead when compared to **DYNACUT**. **DYNACUT** can be used orthogonally with these methods to provide a robust and practical framework to reduce the vulnerabilities of an application.

Chapter 8

Conclusions, Limitations and Future Work

8.1 Conclusion

Existing Debloating and Software Customization Techniques provide frameworks to remove unused code from an application but do not support removing code *dynamically* or at application *runtime*.

We present DynaCut– A framework to dynamically customize a process. DynaCut uses execution traces of the application to identify the unused code in the application. At the application runtime, DynaCut uses a process-rewriting method to *customize* the process – by adding fault handlers, traps, and removing initialization code. We presented the Design and Implementation of a prototype of DynaCut and evaluated it using 9 real-world applications. The evaluation shows that DynaCut can remove up to 56% of the executed code the case of server applications and up to 10% of the application code size is removed with minimal overhead.

DynaCut is a novel approach to edit a process *Dynamically* with minimal overhead. We presented a prototype of an application that can *adapt* a process to changing usage contexts with a tolerable overhead.

DynaCut can also provide security benefits to the application by removing unused code – reducing the viability of code-reuse attacks. Apart from removing unused code, DynaCut allows the end-user to customize the features of the application based on the usage context. Vulnerable features can be blocked and then if the need arises, they can be re-enabled.

8.2 Limitations

DynaCut uses code traces generated using the user-defined test cases and identifies the unused code. But, it does not go beyond these traces to identify code that is *wanted* by the test case but is not identified in the traces. For this, a heuristics-based approach like Razor [37] can be used to identify the basic blocks used for application initialization accurately.

DynaCut currently targets removing code in the application binary only. We did not consider removing unwanted code in the in-memory copy of `libc` for our prototype. Removing unwanted and initialization code in `libc` could reduce the attack surface of the application even further.

Our modifications to CRiU *increases* the CRiU dump size. This is because we dump *all* the code pages of the application and not only those that we modify. For some applications, the dump folder size was up to 6GB, making the CRiU dump unmanageable.

8.3 Future Work

The Future work that can be carried out to improve the current capabilities of DynaCut is highlighted in this section. We also discuss how DynaCut can be used to augment the capabilities of existing debloating and code customization frameworks.

One work can be to optimize the set of CRiU dump images to reduce its size by including only the code pages that need to be modified for the customization in the dump.

Customizing `Libc` and retaining only the functions in `Libc` that are required after the initialization phase of the application can be another direction in which the current prototype can be improved.

Our work can be combined with several existing debloating and customization techniques to provide a robust framework to reduce the attack surface of an application. For example, DynaCut can be combined with existing Dynamic Software Update techniques to patch vulnerable code or also with existing debloating techniques to remove code used at runtime on top of removing the unused code blocks.

A robust security evaluation of DynaCut can also be carried out. Binaries customized using DynaCut can be attacked and evaluations detailing the viability of these attacks can be performed.

Bibliography

[1] 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. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387550. URL https://doi.org/10.1145/3342195.3387550.

[2] APACHEBENCH. ApacheBench - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

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

[4] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sept 2004.

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

[6] The MITRE Corporation. CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028, 2013.

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

[8] Mick de Peinder. x64 Return-to-plt attack. https://mickdepeinder.nl/Posts/x64-return-to-plt, 2020.

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

[10] DynamoRIO. DynamoRIO: Dynamic Instrumentation Tool Platform. https://dynamorio.org/, 2021.

[11] Python Software Foundation. Python. https://www.python.org/, 2021.

[12] Wikimedia Foundation. Dead code. https://en.wikipedia.org/wiki/Dead_code, 2021.

[13] WikiMedia Foundation. Executable and Linkable Format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, 2021.

[14] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1009–1022. ACM, 2019. doi: 10.1145/3319535.3345665. URL https://doi.org/10.1145/3319535.3345665.

[15] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In Manuel Egele and Leyla Bilge, editors, *23rd International Sym-*

posium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020, pages 443–458. USENIX Association, 2020. URL https://www.usenix.org/conference/raid2020/presentation/ghavanmnia.

[16] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1749–1766. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia.

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

[18] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 249–264, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315616. doi: 10.1145/2384616.2384635. URL https://doi.org/10.1145/2384616.2384635.

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 380–394. ACM, 2018. doi: 10.1145/3243734.3243838. URL https://doi.org/10.1145/3243734.3243838.

[20] Gerard J. Holzmann. Code inflation. *IEEE Software*, 32(2):10–13, 2015. doi: 10.1109/MS.2015.40.

[21] M. Frans Kaashoek Jeff Arnold. Ksplice: Automatic Rebootless Kernel Updates. https://www.ksplice.com/doc/ksplice.pdf, 2021.

[22] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In Radu F. Babiceanu, Hélène Waeselynck, Raymond A. Paul, Bojan Cukic, and Jie Xu, editors, *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, pages 122–131. IEEE Computer Society, 2016. doi: 10.1109/HASE.2016.27. URL https://doi.org/10.1109/HASE.2016.27.

[23] Michael Kerrisk. ld.so(8) — Linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html, 2021.

[24] Michael Kerrisk. sigaction(2) — Linux manual page. https://man7.org/linux/man-pages/man2/sigaction.2.html, 2021.

[25] Michael Kerrisk. sigreturn(2) — Linux manual page. https://man7.org/linux/man-pages/man2/sigreturn.2.html, 2021.

[26] Michael Kerrisk. system(3) — Linux manual page. https://man7.org/linux/man-pages/man3/system.3.html, 2021.

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

[28] Osnat Levi. Pin - A Dynamic Binary Instrumentation Tool .

https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2021.

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

[30] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1023–1040, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354218. URL https://doi.org/10.1145/3319535.3354218.

[31] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. Dynamic and secure memory transformation in userspace. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I*, volume 12308 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2020. doi: 10.1007/978-3-030-58951-6_12. URL https://doi.org/10.1007/978-3-030-58951-6_12.

[32] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association. URL https://www.usenix.org/conference/usenix-09/immediate-multi-threaded-dynamic-software-updates-using-stack-reconstruction.

[33] David L Mulnix. Intel MPK. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html, 2019.

[34] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 72–83, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933204. doi: 10.1145/1133981.1133991. URL https://doi.org/10.1145/1133981.1133991.

[35] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 241–254, 2019.

[36] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/park-soyeon.

[37] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1733–1750. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/qian.

[38] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '17, page 65–70, New York, NY, USA, 2017. Association for

Computing Machinery. ISBN 9781450353953. doi: 10.1145/3141235.3141242. URL https://doi.org/10.1145/3141235.3141242.

[39] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. Debloating software through piece-wise compilation and loading. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 869–886. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/quach.

[40] James R. Processor Tracing. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, 2013.

[41] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. Bintrimmer: Towards static binary debloating through abstract interpretation. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 482–501. Springer, 2019. doi: 10.1007/978-3-030-22038-9_23. URL https://doi.org/10.1007/978-3-030-22038-9_23.

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

[43] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020. ISBN 978-1-939133-17-

5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

[44] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375542. doi: 10.1145/3381052.3381326. URL https://doi.org/10.1145/3381052.3381326.

[45] Inc. Synopsys. The Heartbleed Bug. https://heartbleed.com/, 2020.

[46] The Clang Team. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, 2021.

[47] The Clang Team. SafeStack. https://clang.llvm.org/docs/SafeStack.html, 2021.

[48] *Executable and Linkable Format (ELF).* Tool Interface Standards (TIS), May 1995.

[49] REST API Tutorial. HTTP Methods. https://restfulapi.net/http-methods/, 2021.

[50] url:hackingblind. Hacking Blind. http://www.scs.stanford.edu/brop/bittau-brop.pdf.

[51] url:pyelf. pyelftools github. https://github.com/eliben/pyelftools.

[52] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-

9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[53] Veracode. What Is a Buffer Overflow? Learn About Buffer Overrun Vulnerabilities, Exploits & Attacks. https://www.veracode.com/security/buffer-overflow, 2021.

[54] w00d. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[55] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. Secure and efficient in-process monitor (and library) protection with intel mpk. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375238. doi: 10.1145/3380786.3391398. URL https://doi.org/10.1145/3380786.3391398.

[56] Wikimedia. Dynamic software updating. https://en.wikipedia.org/wiki/Dynamic_software_updating, 2020.

[57] Wikimedia. Return-oriented programming. https://en.wikipedia.org/wiki/Return-oriented_programming, 2021.

[58] 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. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*, pages 367–382, 2016.

[59] SengMing Yeoh. Secure and Efficient In-Process Monitor and Multi-Variant Execution. https://vtechworks.lib.vt.edu/bitstream/handle/10919/102158/Yeoh_S_T_2021.pdf?sequence=1&isAllowed=y, 2020.

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

[61] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.