Code Specialization through Dynamic Feature Observation

Priyam Biswas
biswas12@purdue.edu
Purdue University

Nathan Burow
nathan.burow@gmail.com
Purdue University

Mathias Payer
mathias.payer@nebelwelt.net
EPFL

ABSTRACT

Modern software (both programs and libraries) provides large amounts of functionality, vastly exceeding what is needed for a single given task. This additional functionality results in an increased attack surface: first, an attacker can use bugs in the unnecessary functionality to compromise the software, and second, defenses such as control-flow integrity (CFI) rely on conservative analyses that gradually lose precision with growing code size.

Removing unnecessary functionality is challenging as the debloating mechanism must remove as much code as possible, while keeping code required for the program to function. Unfortunately, most software does not come with a formal description of the functionality that it provides, or even a mapping between functionality and code. We therefore require a mechanism that—given a set of representable inputs and configuration parameters—automatically infers the underlying functionality, and discovers all reachable code corresponding to this functionality.

We propose Ancile, a code specialization technique that leverages fuzzing (based on user provided seeds) to discover the code necessary to perform the functionality required by the user. From this, we remove all unnecessary code and tailor indirect control-flow transfers to the minimum necessary for each location, vastly reducing the attack surface. We evaluate Ancile using real-world software known to have a large attack surface, including image libraries and network daemons like nginx. For example, our evaluation shows that Ancile can remove up to 93.66% of indirect call transfer targets and up to 78% of functions in libtiff's *tiffcrop* utility, while still maintaining its original functionality.

ACM Reference Format:
Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code Specialization through Dynamic Feature Observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, USA, 13 pages. https://doi.org/10.1145/3422337.3447844

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CODASPY '21, April 26–28, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8143-7/21/04…$15.00
https://doi.org/10.1145/3422337.3447844

1 INTRODUCTION

Similar to the second law of thermodynamics, (software) complexity continuously increases. Given new applications, libraries grow to include additional functionality. Both applications and libraries become more complex based on user demand for additional functionality. The Linux kernel is an important example of this phenomenon: its code base has grown substantially over the last 35 years (from 176K LoC to 27.8M LoC [10, 11]). Yet, given a single task, only a small subset of a program (or library) is required to be executed at runtime. This increase in code size can also be seen in network facing applications such as nginx or tcpdump, which deal with, e.g., IPv4, IPv6, or proxy settings, as well as image processing libraries, which face increasingly complex file formats as standards expand to support more features. This feature bloat results in a massive amount of unneeded complexity and an ever-growing attack surface. Ideally, applications would be customized with the minimal set of features required by the user, and only the minimum amount of code inlined from imported libraries.

Software complexity results in a flurry of challenges rooted in security, performance, and compatibility concerns. In our opinion, security is the most pressing of these challenges as security flaws can lead to potentially irreversible losses from adversarial exploitation. While functionality may not be required for a given task, adversaries may still find ways to exercise it, increasing the attack surface of a program [12, 19, 32, 33]. Additionally, the precision of popular mitigations such as control-flow integrity (CFI) degrades when more code is introduced. Deployed CFI mechanisms [5] leverage function prototypes to disambiguate the target sets of valid targets. Additional complexity increases the probability that functions with the same signature pollute the same target set.

Removing unnecessary functionality is extremely challenging, as the majority of programs and libraries do not come with a formal description of their functionality. Even worse, there is no clear mapping between functionality (i.e., an exposed API) and the underlying code. Reducing the attack surface and removing unnecessary code requires a mechanism to infer this functionality to code mapping based on an *informal description* of the necessary functionality.

Debloating removes unnecessary code at various levels of granularity [14, 29, 40, 41, 43]. Removing dead code reduces the number of gadgets and (potentially buggy) unreachable functionality. Due to the lack of formal descriptions of functionality, these approaches all remain conservative and must include potentially unneeded functionality. Unfortunately, debloated code may still contain vulnerabilities and sufficient targets for an attacker [16].

Our core idea is to enable the user to select the minimum required functionality (by providing a set of example seeds),

thus establishing an informal description of functionalities in a program. While this approach was previously used to reverse engineer and extract functional components [37], we are the first to leverage user help to *specialize complex software*. The user provides a set of inputs that exercise the required functionality and a configuration of the software (as part of the environment). Our approach, Ancile, then specializes the program in three steps. First, Ancile infers the required functionality and code through *seed demonstrated fuzzing* (fuzzing based on user-provided seeds that exercise the desired functionality). Second, Ancile *removes all unnecessary code* in a compilation pass. Third, Ancile computes *minimal CFI target sets* (based on individual indirect call locations instead of over-approximation on function prototypes) to enforce strong security properties. The third step is essential as a sufficient number of gadgets may remain in the code even after debloating to enable arbitrary code reuse attacks. By generating per-control-flow transfer location target sets, we minimize the reachable locations and increase the precision of fine-grained CFI to a per-location basis, the highest possible precision for a static CFI approach. Type-based CFI (the current standard) uses one target set per function type, i.e., all indirect call locations with the same call signature allow the same set of targets. Ancile specializes this set for each indirect call location to only targets that are observed during the analysis phase, strictly increasing the precision. Improving precision via specialization comes with the challenges of introducing false positives and false negatives, we discuss those Section 3.

Note that, we propose fuzzing not primarily as a bug finding tool (although Ancile may discover bugs during focused fuzzing that can be reported to the developer) but as a tool for analyzing exercised code. Coverage-guided greybox fuzzing uses code coverage as a feedback to map code to inputs. We use this insight to discover the exercised functionality and to map the corresponding code to user-selected inputs. The contributions of our approach are below:

- We design a code specialization technique that repurposes fuzzing to reduce a program to the minimal amount of code required for a given functionality. Our technique not only removes unnecessary code, but also specializes control-flow checks by creating a reduced target set.
- We present a comprehensive analysis of Ancile on real-world applications to show the effectiveness of fuzzing as a way to generate precise path information.

2 BACKGROUND

We provide a brief introduction of debloating and CFI, both of which seek to minimize the attack surface of applications. We also describe fuzzing and sanitization as these concepts are integral to our approach.

2.1 Attack Surface Debloating

As software has expanded its functionality to serve more users, the size and complexity of libraries and applications has grown dramatically over time, resulting in software bloat. For example, a recent study showed that most applications only use 5% of *libc* [41]. Code bloat imposes a cost: increased attack surface for adversaries. Software debloating is a technique that helps prune the program's attack surface by removing extraneous code. Several approaches have been proposed such as debloating via reinforcement learning [29] or trimming unused methods [35]. However, trimming unused or rarely used features cannot alone prevent Control-Flow Hijacking (CFH). By manipulating remaining indirect call sites, an attacker can still launch code-reuse attacks.

Code debloating improves security along two dimensions: code-reuse reduction and bug reduction. First, code debloating reduces the amount of available code, making it harder for an attacker to find gadgets for a code-reuse attack. Second, feature based code debloating approaches reduce attack surface by removing potentially reachable buggy functionality, making it harder for the attacker to find an exploitable bug. Unfortunately, security effectiveness of existing code debloating is inherently limited by *the amount of code that remains*. Any functionality in the program requires code, and even tiny programs [30] provide enough code for full code-reuse attacks. While code debloating may be effective in removing some reachable bugs, it is not effective in stopping code-reuse attacks as any remaining code will be sufficient for such attacks.

2.2 Control-Flow Integrity

Another prominent mechanism for reducing attack surface is Control-Flow Integrity (CFI), the state-of-the-art policy for preventing code-reuse attacks in C and C++ programs. Its key insight is that to perform a control-flow hijacking attack, attackers must modify the code pointer used for an indirect control-flow transfer (direct control-flow transfers are protected as the target is encoded in read-only code). CFI builds, at compile time, a set of legitimate targets for each indirect and virtual call, and, at runtime, validates that the observed target is in the allowed set. By verifying the target, CFI prevents the use of any corrupted code pointer.

In contrast to debloating which restricts attack surface by removing unneeded code, CFI does so by allowing only valid targets for each indirect control-flow transfer. In other words, *CFI removes extraneous targets from indirect branches*. Code debloating by itself, i.e., without CFI, may remove large amounts of executable code but the remaining code could still be used for arbitrary code reuse attacks. An adversary only needs a single usable target but a defense must prohibit all reachable targets to be effective. Partial target reduction through debloating is insufficient to stop an attack.

State-of-the-art CFI mechanisms [17] focus on a conservative static analysis for building target sets, including more targets than just the valid ones. While this approach has no false positives, it over-approximates the targets. It is also possible to use dynamic analysis to construct the target sets, potentially introducing false positives, but greatly improving the precision of the analysis. Here, we discuss both analysis techniques and their trade-offs.

Static Analysis-Based CFI. Static analysis-based CFI mechanisms compute the allowed target sets at compile time. The

analysis discovers the set of functions that the programmer intends to target at a given indirect call site. In compiler terms, the analysis is looking for every reaching definition of the function pointer used at the indirect call site. Implementations of the analysis quickly run into the alias analysis problem, and so have to fall back to more tractable, albeit over-approximate, techniques. Early mechanisms reverted to allowing any address taken function [13] to be targeted at any indirect call site. Subsequent mechanisms improved this to any function with a matching prototype [52]. Recent work has even looked at using a context-sensitive and flow-sensitive analysis to further limit the target sets [25, 26]. While such works increase the precision of the analysis, aliasing prevents achieving full sensitivity.

Dynamic CFI. Compared to static CFI, dynamic CFI mechanisms modify the target sets of the control-flow transfers during the execution of the program. Dynamic CFI is generally more precise than static CFI as it starts off with a static target sets but then uses runtime information to further constrain the target sets.

Several works have leveraged hardware support to restrict the target sets during runtime. πCFI [38] begins with an empty control-flow graph and activates control transfers as required by specific inputs. However, this approach does not execute any address deactivation which may degenerate to the full static control-flow graph (CFG). PathArmor [53] takes advantage of hardware support, specifically the 16 Last Branch Record (LBR) registers to effectively monitor per-thread control-flow transfers. It limits the verification process to only security critical functions, and verifies the path to these critical functions by using a path cache. PittyPat [21] improves on this by collecting runtime traces via Intel PT, and verifies them in a separate process, halting execution at system calls to synchronize with the verification process. While it is precise (assuming the entire execution is traced), PittyPat consumes significant additional resources, e.g., another core for the verification process. μCFI [31] improves PittyPat by recording execution contexts using Intel PT, and observing unique code targets for each indirect control-flow transfer. As PittyPat, it relies on a separate monitoring process.

CFI Security. CFI does not protect against data-only attacks. An attacker that compromises the data of a process can bend execution [19, 32, 33] to any allowed functionality and, if a path in the original CFG exists, CFI will allow execution of that path. While CFI limits code execution to legitimate targets under some execution of the program, it does not remove unneeded functionality.

CFI prohibits rogue control flow to unintended locations while code debloating removes unnecessary code. In combination, CFI and code debloating can reduce the exposure of a program but are limited by the remaining code as both approaches are conservative, resulting in an over-approximation of the required functionality.

2.3 Fuzzing

Fuzzing [1] automatically generates test cases. Coverage-based fuzzers such as American Fuzzy Lop (AFL) [2] create a new test case by mutating interesting inputs that trigger new code paths. Their mutation based strategy leads them to test many inputs that cover the same code paths, causing them to explore the possible data-flows of the application as well. Fuzzers operate from a seed input, mutating it in their search for new code-paths while simultaneously exploring data paths as a result of their search.

Ancile relies on extensive path coverage to generate comprehensive target sets for indirect call-transfers of the selected functionality. Guided fuzzing [6] facilitates finding new code paths from an indirect call site. With the knowledge of deeper path information, target discovery has become more efficient.

2.4 Sanitization

Sanitization is a dynamic testing technique that effectively detects policy violations at runtime [50]. A sanitizer generally instruments the program during compilation to enforce some security policy. The instrumentation collects metadata about the program execution and continuously checks if the underlying policy is violated.

AddressSanitizer (ASan) [34, 48] employs a specialized memory allocator, and instruments memory accesses at compile time to detect out-of-bounds accesses to heap, stack, and global objects, as well as temporal bugs. ASan is a tripwire-based approach that creates redzones, and checks each memory access to detect memory safety violations. Fuzzing then triggers memory access bugs and ASan detects them. Other well-known sanitizers are memory Sanitizer (MSAN) [51] to detect accesses to uninitialized memory, UndefinedBehaviorSanitizer (UBSan) [9] to catch various types of undefined behavior, or LowFat [22, 23] to detect out of bounds accesses efficiently.

As Ancile uses fuzzing for functionality inference, we must distinguish between correct functionality and potential bugs. To avoid memory corruption bugs in our allowed functionality, we compile target programs with ASan during the inference phase. Ancile ensures all the explored targets via fuzzing are indeed valid targets.

3 CHALLENGES AND TRADE-OFFS

Code specialization is a technique used to generate more efficient code for a specific purpose from generic code [36]. The core issue of code specialization is the prediction of effective code-behavior in order to generate precise control-flows. Specializing an application allows us to apply both attack surface reduction techniques at once, by removing code unused by the deployment scenario, *and* restricting targets to exactly the purposefully valid sets. However, automatically specializing code to only support a user specified configuration is challenging. Static analysis quickly degenerates to the aliasing problem [42], and has difficulty determining if a function is required for a particular functionality. Dynamic analysis is an attractive alternative, however, it requires that all valid code and data paths for a particular configuration are explored.

Dynamic analysis has been made practical by recent advances in automatic testing, and in particular coverage-guided fuzzing [2, 6, 39, 44]. Given a minimal set of seeds that cover the

desired behavior, fuzzers are capable of quickly and effectively exploring sufficient code and data paths through a program to observe the required indirect control-flow transfers for a given configuration. CFI target sets are then restricted to the observed targets for the desired functionality of the application, e.g., an IPv4 deployment of nginx with no proxy. Note that the dynamic analysis can occur offline, with only traditional CFI set checks, which incur minimal performance overhead, required at run time. *Ancile leverages fuzzing to correlate functionality with code.* Fuzzing's code exploration serves as a mapping process from functionalities to relevant code-regions. The coverage information from fuzzing enables us to effectively specialize software by replacing conservative analysis of valid cases with a more precise analysis of what states are reachable in practice. The correctness of the specialization procedure depends on successfully mapping functionality to functions. Ancile maps code at function level granularity. When executing the program with the desired functionality, Ancile marks and includes all executed functions and prunes functions that are not exercised. Hence, mitigating challenges of partial code removal.

However, using fuzzing as a path exploration technique introduces its own set of challenges: (i) generating a dynamic control-flow graph (CFG) for user-selected functionality, (ii) projection of dynamic CFG in functionality-based debloating, (iii) precision vs soundness in CFI target analysis, and (iv) the risk of introducing false positives/negatives due to inherent randomness in fuzzing. We now discuss each of these challenges in turn and how we address them.

Challenge i. Generating a dynamic CFG. Given a program with a set of functionalities f_1,f_2,f_3,...,f_n and a user-specified functionality $f_s \subset \{f_1,f_2,f_3,...,f_n\}$, we must discover the code required by that particular functionality, f_s. For example, a user may only require the *tiffcrop* functionality from the image library libtiff. To generate a dynamic CFG for a given functionality, we need to explore all required and valid control-flows exercised by that functionality within the program. In the CFG, we need to include only the targets originated from exercising the desired functionality and exclude any potential targets that are not relevant. Hence, if a control-flow has a target set S $= t_1$,t_2,t_3,...,t_x and the CFG should have only the subset of the target set $S' = t_1$, t_3,... coming from the desired functionality. We also have to ensure that S - S' targets are not included in the dynamic CFG. Ancile addresses this challenge by taking as input a set of seeds and configuration demonstrating the required functionality (f_s), and then uses these to fuzz the application in order to retrieve the relevant control flows and record the valid targets. For instance, if the desired functionality of the application *tcpdump* is to read only *pcap* files, the user needs to provide *pcap* files as seed. Ancile starts with an empty CFG and adds edges in the dynamic CFG only if their execution is observed in the set of valid executions of reading *pcap* files.

Challenge ii. Projection of dynamically generated CFG in functionality-based debloating. To prune unneeded functionality, we need to map the control-flow information into relevant code. In order to do so, we guide fuzzing by carefully selecting inputs to explore the intended functionality. Similar to Razor [40] and binary control-flow trimming [28], Ancile utilizes test cases to trace execution paths. Ancile also takes advantage of the power of coverage-guided fuzzing to explore deeper code paths pertinent to the desired functionality. To ensure that the fuzzed functionality has covered all possible paths, we evaluate the targeted utility with a different set of test cases. Ancile then removes any not-executed functions.

Challenge iii. Precision vs soundness. Ancile trades soundness for precision when constructing CFI target sets. State-of-the-art CFI mechanisms have focused on a conservative static analysis for building the CFG, resulting in a conservative over-approximation of indirect control-flow targets. These CFI mechanisms quickly run into the alias analysis problem, and so must fall back to more tractable, albeit over-approximate, techniques. Recent approaches have looked at using context-sensitive and flow-sensitive analyses to further limit the target sets [25, 26]. While such works increase the precision of the analysis, aliasing prevents achieving full sensitivity.

It is also possible to use dynamic analysis to construct target sets, potentially introducing false positives, but greatly improving the precision of the analysis. Several works [21, 38, 53] introduce hardware requirements to restrict the target sets during runtime. Both static and dynamic approaches inherently over-approximate as existing CFI solutions are oblivious to a minimal, user-specified functionality. Static analysis-based approaches leverage only information available during compilation, while dynamic analysis-based approaches use runtime information to further constrain the target sets. Still, existing dynamic mechanisms result in over-approximation in the target set. Ancile extensively fuzzes the desired functionality to infer the required control-flow transfers. Fuzzing's efficiency comes from its fundamental design decision: to embrace randomness and practical results rather than theoretical soundness. Consequently, fuzzing gives no guarantees about covering all possible code or data paths, but covers them well in practice.

Challenge iv. False positives and false negatives. Our goal is to minimize the number of targets for individual CFI checks. Ancile restricts per-location CFI targets by combining per-function removal along with CFI-based target removal.

An unintended function included in the target set is a false negative. This can happen in two scenarios, (i) a fuzzing campaign performing invalid executions; and (ii) exploring traces outside of the desired functionality. Ancile guarantees valid executions by using Address Sanitizer (ASan) along with fuzzing. By selecting relevant seeds, we prime fuzzing to only explore relevant code regions. For example, when using a PNG as seed, we are much more likely to explore PNG code features than to mutate the seed into a valid JPG image. To improve the confidence in our discovered targets, we use two separate fuzzers to cross check all discovered targets. We have tuned our fuzzing campaign timeline and observed that neither increasing the duration beyond 24 hours nor repeating the fuzzing campaign multiple times does discover more targets. We also performed manual static analysis with 40 test cases to verify these results.

False positives occur if valid and intended targets are not included in the generated set through lack of fuzzing coverage. Ancile starts with the minimum set of seeds that exercise the intended functionalities, giving a lower-bound of targets. Next,

fuzzing discovers targets that were not previously included. Moreover, to increase confidence in the discovered target set, we repeat each fuzzing campaign multiple times. We discuss false positives/negatives in Section 6.

4 ANCILE DESIGN

Based on the user-selected functionality (through provided seeds), Ancile generates specialized binaries. The design of Ancile is motivated by the need for precise control-flow information so that this information can be used to debloat the target program, *reducing its exposed attack surface*. The user *informally specifies* the desired functionality by providing seed inputs that explore that functionality. Ancile operates in three distinct phases, as shown in Figure 1. First, Ancile performs targeted fuzzing (using the seeds provided by the user) to infer the CFG and to explore code associated with the required functionality (including error paths). This step infers all of the necessary information for the next two steps. Second, Ancile removes any unnecessary code using a compiler pass, reducing the program's attack surface. Third, Ancile leverages the precise CFG to customize CFI enforcement to the observed CFG. This customization increases the precision of CFI to only observed targets. These observations result in the following requirements:

Desired Functionality. Every application has its own set of features. By desired functionality, we mean one or more features of the application that the user intends to exercise. For example, in tcpdump, the user may only want to exercise the feature that reads `pcap` files.

Seed Selection. The minimum number of inputs required to exercise the desired functionalities is selected. For example, to exercise the feature of reading a `pcap` file, the user only needs to provide a captured `pcap` file.

User Involvement. Ancile requires two sets of input from the user, (i) necessary command line arguments to select the functionality; and (ii) a minimum set of seeds that exercise this functionality. For reading a `pcap` file, the user must provide (i) the `-r` command-line argument, and (ii) a `pcap` file as an input seed. Ancile results in low user burden. The user only needs to specify input configurations for their chosen functionality. The user does not have to carry out any form of source code annotations or code changes and does not require any knowledge of the source code. Applying Ancile is as easy as building a software package which any package maintainer or advanced user can do. It is a one-time effort to generate the specialized application.

The key insight of Ancile is the functionality analysis. It is this analysis which allows us to *automatically* specialize an application, simultaneously removing extraneous features and shrinking the attack surface by restricting the set of allowed indirect control-flow transfers. Selection of the required functionality depends on the type of application as well as user requirements. Ancile minimizes the user burden for feature selection. For example, if a user wants to read `pcap` files using tcpdump, she will configure Ancile to execute tcpdump with the command line option `-r`, and a sample `pcap` file as input. Ancile also takes advantage of existing unit test-suites that comes with the application package to exercise functionality.

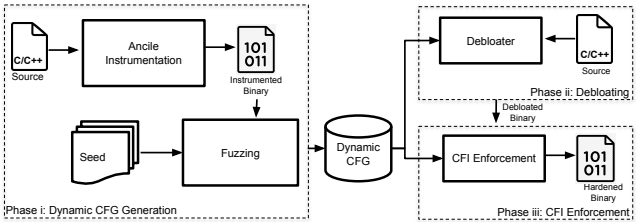


Figure 1: Ancile operates in three phases: (i) Dynamic CFG Generation (record control flow), (ii) Debloating (remove unnecessary functionality), and (iii) CFI Target Analysis.

Ancile uses fuzzing to infer the code covered by an informally-selected functionality. Input seeds are used to exercise the desired functionality. Coverage-based fuzzing excels at finding code paths from a given seed. For each target in our per CFI-location target sets, fuzzing produces an execution that witnesses that specific target. The challenge becomes ensuring that the set of executions used by our functionality analysis fully covers the control and data flows of the desired functionality. We show that fuzzing, in conjunction with a small set of test cases that observe the desired functionality, can be leveraged to generate a precise CFG.

Ancile then utilizes the dynamic CFG constructed in the dynamic CFG generation phase as a mechanism for (i) debloat unnecessary code and (ii) tighten CFI checks to restrict indirect control-flow to a set of targets required by a given user specification. Ancile can achieve the best possible precision with negligible runtime overhead, i.e., set checks inserted at compile time. Therefore, we believe that increased specialization is the way of the future for "prevent-the-exploit" defenses.

4.1 Dynamic CFG Generation

Ancile requires the user to select the desired functionality of the program by providing corresponding input. These input seeds can come from, e.g., unit tests, examples, or be custom tailored by the user. For example, the network sniffer *tcpdump* offers a variety of features, from directly capturing network packets to processing recorded traces. A user may want to only process recorded traces of a single protocol. Building off this informal specification, Ancile performs dynamic fuzzing that identifies (i) all the executed functions, and (ii) the targets of indirect function calls. Any function that has not been observed via direct or indirect calls during this phase is considered extraneous and hence, is not included in the CFG. At this point, our analysis is fully context and flow sensitive, as it directly depends on actual executions.

After this analysis, the observed targets are aggregated over each indirect call site. This aggregation results in some over-approximation and a loss of full context and data sensitivity. However, every target we allow is valid for *some* execution trace, which is a significantly stronger guarantee than is provided by static analysis-based CFI [17]. Static analysis-based target sets only guarantee that every target *may* be required by an execution trace. Put another way, our dynamic analysis

recovers the *programmer-intended target sets*, rather than an over-approximation thereof.

Ancile recompiles the application with not only the coverage instrumentation for grey box fuzzing, but also to log the targets for direct and indirect control-flow transfers. In particular, we cover forward edges, leaving return edges for more precise solutions such as a shadow stack [18]. Ancile rejects all executions that AddressSanitizer [46] flags as an error to ensure that all observed executions are in fact valid.

As fuzzing is incomplete, the core risk of this approach is that some required functionality is not discovered and therefore unintentionally removed. Our analysis could potentially introduce false positives (prohibiting valid indirect control-flow transfers). This is in direct opposition to the conservative approach employed by static analysis, which over-approximates and thus weakens security guarantees. In contrast, Ancile only allows the targets for a particular functionality.

The increased security guarantees through this specialization provide a new avenue for the security community to explore. Our evaluation Section 6 shows that with the increasing power of automated testing techniques such as fuzzing [2], robust test sets maintained by many projects [7, 8], and a wealth of prior work on sanitizers [46] to validate execution traces, Ancile did not cause false positives in our observed test cases.

4.2 Debloating Mechanism

In automatic code specialization, unneeded code is discarded and the debloated program contains only the required functionality. Given the user's functionality selection, the challenge of debloating comes from mapping functionality to code regions. One possible approach to address this challenge is to learn code regions through valid program executions that exercise the desired functionality. In other words, we require a set of inputs that exercises, at least minimally, all desired functionality.

By taking advantage of the dynamic functionality observation performed in the first phase of our analysis, Ancile discovers all reachable and executable code. This code analysis can be considered a simple marking phase that records all reachable code. Based on the recorded execution traces, Ancile removes all unneeded code. As a second compilation pass, with the marked code from the fuzzing campaigns, we then tailor and remove all unnecessary code on a per function basis. All functions that are unreachable are replaced with a single empty stub. If this stub is reached, the program is terminated with an error message.

Ancile debloats at function level granularity. Debloating at function level (compared to the more fine-grained basic block level) ensures that each function includes all necessary code. A function invocation during the analysis phase indicates that the function is part of the user-selected functionality. While debloating at the basic block level would be more precise, ensuring that the fuzzer executes all possible paths through each function is currently an unsolved open challenge, we therefore opted for function level granularity. Hence, the correctness of Ancile's debloating mechanism depends on the exercise of only the required functions pertaining to the specified functionality. Seed demonstrated fuzzing along with proper sanitization can ensure the execution of the necessary functions.

Ancile keeps only the functions that are observed during the analysis phase. Depending on the underlying platform and instruction set, exceptions are implemented differently. On Linux/x86-64 exceptions are using the Itanium ABI and come with a zero cost abstraction on top of DWARF4. We include the reachable exception handlers in the final binary, as they can be triggered through DWARF4-based exception handling.

4.3 CFI Target Analysis

Although, debloating restricts a program's attack surface by removing unneeded code, it is still possible that vulnerabilities remain in non-bloated code. To ensure tighter security in the specialized binary, Ancile removes extraneous targets from indirect control-flow transfers in the remaining code.

The main goal of Ancile's CFI target analysis is to achieve minimal target sets for indirect branches. It does so by only allowing targets that are required for the specified functionality *and actually observed at runtime*. For each target, we ensure that there is at least one dynamic witness, i.e., a valid execution trace that includes the indirect call. Hence, Ancile solves the aliasing problem of static analysis based approaches and increases precision.

Based on the inferred CFG that is tied to the actual execution of the desired behavior, Ancile learns—for each indirect control-flow transfer—the exact set of targets observed during execution. This set is strictly smaller than the set of all functions with the same prototype. Once the target sets are created, we recompile the application to a *specialized* form, which enforces the target sets derived from our functionality analysis.

Since we focus on static CFI enforcement mechanisms, deciding if a target is allowed depends purely on the information known at compile time, regardless of how that information was obtained. For example, if two paths in a program result in two different targets at a location then the most precise static mechanism will always allow both targets (as it cannot distinguish the runtime path without tracking runtime information). In contrast, dynamic enforcement mechanisms can modify the target sets depending on runtime information (e.g., data-flow tracking). Unfortunately, dynamic mechanisms result in additional runtime overhead (e.g., to update the target sets), increased complexity (for ensuring that the target sets remain in sync), and compatibility issues (e.g., the runtime metadata for the CFI mechanism must be protected against an adversary during the updates). For as long as no hardware extension exists for protecting metadata (e.g., to protect attacker-controlled arbitrary writes from the buggy program), realistically deployable CFI mechanisms will remain static.

5 IMPLEMENTATION

Ancile is implemented on top of the LLVM compiler framework, version 7.0.0. The LLVM-CFI framework has entered mass deployment [15, 20], and its set checks are highly optimized. Consequently, building on top of LLVM-CFI guarantees that our enforcement scheme is efficient, and ready for wide-spread

adoption. As mentioned in the design, the Ancile implementation constitutes three parts: (i) dynamic CFG generation, (ii) debloating and (ii) CFI enforcement, following the description in Section 4.

Dynamic CFG Generation. This functionality analysis phase is implemented as a combination of an LLVM compiler pass and a runtime library. Our instrumentation takes place right after the clang front-end and modifies the LLVM IR code. Ancile is enabled by specifying our new `fsanitize=Ancile` flag.

C/C++ source files are first passed to the clang front-end. The compiler pass adds instrumentation to log all indirect calls and their targets. At the IR level, Ancile adds a call to the logging function in our runtime library before every indirect call. The logging function takes two arguments: location of the indirect call in the source, as well as the address of the targeted function. Additionally, the pass logs all the address taken functions to facilitate the remapping of the logged target addresses to corresponding functions. The runtime library of Ancile generates a hash map to store target set information per call site. To remove extraneous code, Ancile collects information during profiling about function invocations via direct control-flow transfers. This procedure follows the same mechanism described above for indirect control-flow transfers. Hence, Ancile generates a dynamic CFG accommodating all the observed control flows that reflect the user specified functionality.

The challenge associated with fuzzing is to guarantee that paths taken during fuzzing are valid code and data paths. To address such challenges, we leverage AddressSanitizer (ASan) [48], a widely-used sanitizer that detects memory corruptions (e.g., use-after-free or out-of-bounds access). Only non-crashing executions are recorded. Hence, Ancile ensures all the recorded control-flow transfers are from valid execution traces and generates the dynamic CFG.

Debloating. To prune unnecessary code, Ancile utilizes the dynamic CFG to construct the list of observed functions. It then removes any functions that are not in our observed white list, thereby ensuring a custom binary incorporating only the user specified features. It relies on a compiler pass to remove any unintended function.

CFI Mechanism. Ancile enforces the strict targets for the indirect calls based on the dynamic CFG. Despite relying on dynamic profiling, Ancile still enforces target sets statically (i.e., relying only on information available at compile time to embed the target sets in the binary). We have customized LLVM-CFI to adopt Ancile's strict target set at each individual indirect control transfer check points. Our target-set sizes are smaller in most cases and equal to the size of the LLVM analysis in the worst case. In contrast to Ancile, vanilla LLVM-CFI relies on static analysis for target generation and thus fails to solve aliasing, resulting in an over-approximate target sets. The main advantage behind adapting LLVM-CFI is that it is highly optimized and incurs only 1% overhead [5]. Our framework for using LLVM-CFI to enforce user-specified target sets will help the research community to advance control-flow hijacking mitigation by serving as an enforcement API for any analysis that generates target sets.

Note that, Ancile executes its analysis phase during software compilation. No (additional) analysis is required during runtime, i.e., when the program is executed. Hence, the customization procedure is an one-time effort during compilation.

6 EVALUATION

Our evaluation of Ancile focuses on the following research questions:

RQ1. Can fuzzing be used to enable debloating?
RQ2. Can fuzzing be used as a CFI target generator?
RQ3. How can we analyze the correctness?
RQ4. How performant is Ancile (compared to LLVM-CFI)?

For our evaluation, we compared Ancile to vanilla LLVM-CFI. Unlike other debloating mechanisms [40], Ancile performs function level debloating and reduces attack surface by including only the target sets of the indirect control-flows pertaining to the specified functionalities. Hence, we chose to compare to LLVM-CFI which is state-of-the-art CFI mechanism.

We investigated commonly attacked diverse software that offers rich opportunities for customization and specialization. We evaluated Ancile with two popular, and frequently attacked, image libraries `libtiff` and `libpng`, as well as two network facing applications, `nginx` and `tcpdump` which deal with different proxy settings for our analysis. To show the impact of feature selection, we investigated four different cases for each of the applications. We analyzed Ancile with each of the application's standard test-suite (included in the package), as well as two user-selected functionality sets and then compared with vanilla LLVM-CFI.

For the two image libraries, we used the utilities *tiffcrop*, *tiff2pdf* for libtiff and *pngfix*, *timepng* for libpng. We used a set of *tif* and *png* files as input seeds to fuzz the libraries respectively. For tcpdump, we leveraged two sets of command line arguments `-r` and `-ee -vv -nnr` as well as network capture files in the `cap` and `pcap` formats as input seeds. For nginx, we used methods such as GET, POST, and TRACE operations as inputs along with two different configuration settings.

6.1 RQ1: Fuzzing as a debloating tool (RQ1)

With the advancement of efficient coverage-guided mechanisms, fuzzers can be used to observe valid code executions. Ancile learns valid targets yielding from valid execution paths. Ancile utilizes mutational fuzzing via AFL and honggfuzz to explore relevant code paths. To generate complete observed function sets for a desired functionality, it is possible to carefully select input seeds for that particular functionality. For instance, if the user only wants to read `pcap` files via tcpdump, we can provide only `pcap` files as seed. In the case, where the user wants to read both `cap` and `pcap` files, we can then use both type of files as seeds.

In the following sections, we have analyzed fuzzing's effectiveness in debloating and CFI checks. Fuzzing has been mainly used as a bug finding mechanism. To demonstrate its effectiveness as a debloating mechanism, we evaluate code reduction by Ancile on our case studies. Additionally, Ancile

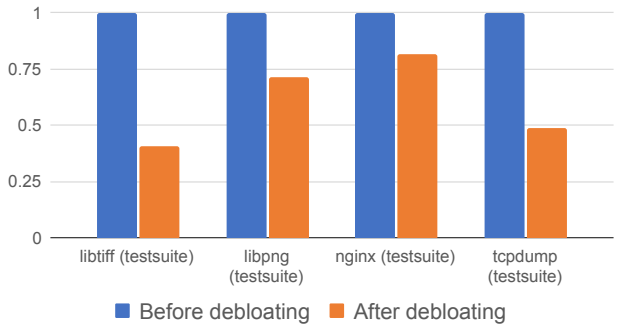


Figure 2: Comparison of the number of functions before and after debloating across libtiff, libpng, tcpdump, and nginx. We used the standard test-suite for each of these applications. Ancile reduces more functions in specialized cases.

Table 1: Sensitive function analysis: Number of indirection level to the sensitive functions from functions present in the target sets of LLVM-CFI and Ancile.

Benchmark	Function		ind. +1	ind. + 2	ind. + 3
libpng	`memcpy`	LLVM-CFI	5	20	17
		Ancile	3	0	0
nginx	`execve`	LLVM-CFI	1	0	0
		Ancile	0	0	0
	`memcpy`	LLVM-CFI	1271	2276	2869
		Ancile	167	272	352
	`mmap`	LLVM-CFI	0	2	4
		Ancile	0	1	1
libtiff	`memcpy`	LLVM-CFI	59	95	66
		Ancile	14	14	11
	`mmap`	LLVM-CFI	1	0	0
		Ancile	1	0	0
tcpdump	`memcpy`	LLVM-CFI	156	670	678
		Ancile	34	22	26

improves the security of the debloated binary by pruning gadgets as well as security-sensitive functions. All performance measurement were done on Ubuntu 18.04 LTS system with 32GB memory and Intel Core i7-7700 processor.

Function Debloating. Ancile debloats applications by removing all unused functions, i.e., code that was never executed during our functionality inference phase. It generates a white list of functions based on the context of the user-specified functionality and removes functions that were not invoked during execution. Figure 2 compares the number of functions before and after debloating is performed across different benchmarks. Additionally, function reduction depends on the specified functionality. Ancile removes around 60% functions for libtiff standard test-suite that comes with the library, where as for a more specialized scenario, for example in case of *tiffcrop* utility, Ancile removes 78% of functions.

Pruning Security Sensitive Functions. The main goal of Ancile is to allow the minimum set of control-flow transfers for the required functionality, thereby minimizing the available attack surface. Sensitive functions belonging to a target set increase the attack surface. We measure if sensitive functions are reachable from (i) indirect calls i.e., they are in the target sets, (ii) at distance-1 (indirection +1), i.e., if a function in the target set calls a sensitive function, (iii) at distance-2 (indirection +2), i.e., if a function in the target set calls a function that calls a sensitive function, and (iv) similarly at distance-3 (indirection +3). In short, we have observed different level of indirect calls in the evaluated benchmarks. We considered `execve`, `mmap`, `memcpy`, and `system` as the set of sensitive functions in our analysis. The main reason behind selecting such functions as sensitive is that an attacker can modify the arguments of these functions such as `system` to execute unwanted actions and gain control of the system. Since, there were no security sensitive function directly in the target set, we exclude criterion (i) from our analysis.

Table 1 shows reachability to sensitive functions from an indirect call site through a sequence of intermediate calls. For instance, in libpng several calls are made to the sensitive function `memcpy`. At indirection+1, indirection+2, and indirection+3 level, there are five, 20, and 17 reachable calls respectively in LLVM-CFI. Ancile restricts these calls to three locations at indirection+1 and in rest of the two cases there are no indirect call sequences to `memcpy`. We have observed another interesting case in nginx, where `execve`, a highly sensitive function, is reachable in indirection+1 in LLVM-CFI, however, Ancile does not allow this call. This call is only made in one rarely-used feature (to hot restart nginx without losing connections when the underlying binary is replaced with a newer version). This demonstrates that focusing on control-flow transfers based on functionality reduces the attack surface when such features are restricted.

Case Study: Gadget Reduction. To better understand the significance of Ancile, we performed a case-study on gadget discovery. We focused on two metrics: (i) Jump Oriented Programming (JOP) gadgets, and (ii) unintended indirect-call gadgets. We did not consider ROP gadgets since our framework is aimed for securing forward edges only and CET [3]-like technology will secure backward edges. We built two versions of nginx: one with LLVM-CFI enforcement and the other with Ancile enforcement along all the unit test-suite features. Using a gadget-discovery algorithm and manual analysis, we observed a 54% reduction in JOP gadgets and a 44% reduction of unintended indirect-call gadgets. This case study shows us that Ancile can indeed help in reducing the number of gadgets in an application.

6.2 RQ2: Effectiveness of fuzzing for CFI

To show the effectiveness of fuzzing as a CFI analysis tool, our aim is to establish that fuzzing is effective in producing *drastically* smaller target sets for indirect control-transfers than previous approaches. We found that Ancile can reduce target sets by 93.66% and 97.94% for the *tiffcrop*, *tiff2pdf* utilities from the libtiff image library. Target set reduction reduces the attack surface, increasing the security of our customized binaries. Any additional target which is not intended to be taken during valid program execution potentially increases an attacker's capabilities. We compare Ancile's target set per call site with LLVM-CFI on libtiff-4.0.9, libpng-1.6.35, nginx-1.15.2

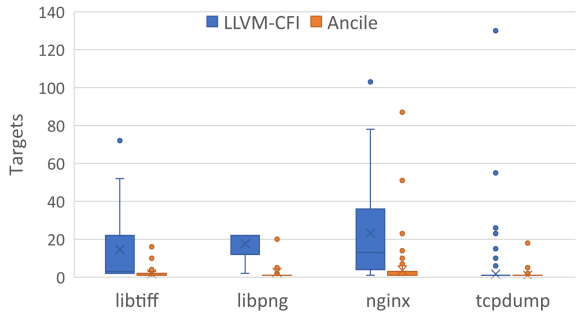


Figure 3: Mean and std. deviation of target sets across our test-suite for LLVM-CFI and Ancile. LLVM-CFI has more callsite outliers with large target sets than Ancile.

and tcpdump-4.9.0, as well as the SPEC CPU2006 benchmark suite. We compare Ancile to LLVM-CFI (the state of the art static CFI mechanism). LLVM-CFI is used, e.g., on Android and Chrome. Ancile generates a customized minimal target set per control-flow transfer in contrast to an over-approximated set generated by static approaches like LLVM-CFI without sacrificing the performance.

To understand the differences in target set generation from different feature selections, we have analyzed the target applications with different user specifications and input seeds. Varying the input seeds for a given specification allows us to examine the effect of path exploration during fuzzing on target set generation.

Figure 3 shows the mean and standard deviation of target set per call site across the four benchmarks for Ancile and LLVM-CFI. We leverage the application's standard test-suite for Ancile's functionality analysis. In each of the benchmarks libtiff, libpng, nginx and tcpdump, LLVM-CFI has on average 73% more targets than Ancile. Furthermore, LLVM-CFI has outliers of call sites with very large target sets. For example, tcpdump has 48 call sites for which LLVM-CFI reports 130 targets, whereas Ancile observes none to at most two targets. To support our claim in target reduction, Table 2 shows the comparison between LLVM-CFI and Ancile for the maximum target set size for each of the benchmarks. This highlights the power of functionality analysis in reducing the attack surface available to attackers.

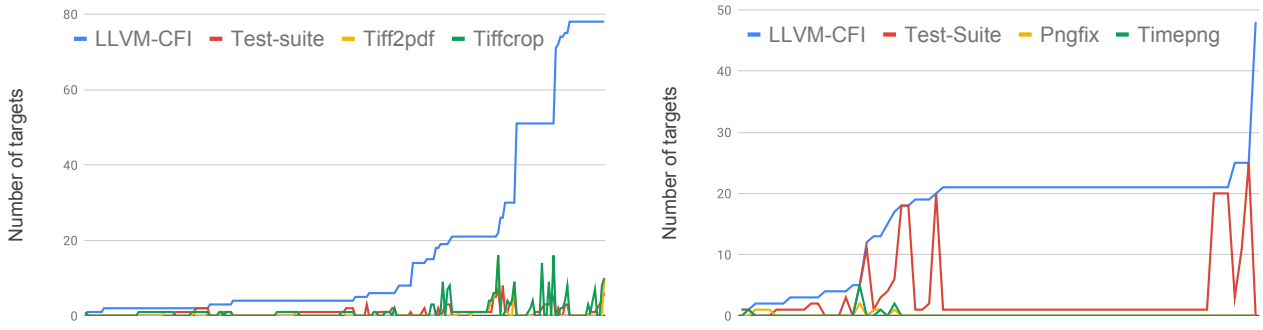
Figure 4 shows the comparison of target-set size per call site between LLVM-CFI and Ancile specializing on different functionalities. In each of the cases, we analyzed target sets obtained from the unit test-suite as well as target sets obtained from the specialization of certain features as mentioned in Section 6. As expected, Ancile reduces the target set sizes for *all* targets, compared to LLVM-CFI. Additionally, fuzzing a particular utility can lead to discovering more targets than the unit test-suite. For instance, for certain indirect control-flow transfers, we observed more targets while fuzzing *tiffcrop* than just running the test-suite.

SPEC CPU2006. In addition to our real-world applications, we also evaluate our prototype on the SPEC CPU2006 benchmark-suite. Working with SPEC CPU2006 enables us to compare with LLVM-CFI. Furthermore, SPEC CPU2006 is the standard performance benchmark, so we included our analysis results for completeness. We used the smaller `test` SPEC benchmark configuration as our functionality specification, and ran the benchmarks once without fuzzing. These target sets were then used to specialize the binaries, and we verified they run with larger `ref` data set, see Section 6.4.

Figure 5 shows the comparison of Ancile, and LLVM-CFI on two SPEC CPU2006 benchmarks, namely 400.perlbench, and 445.gobmk. We chose to focus on these benchmarks as they have the largest number of indirect call sites. We show the cumulative distribution function (CDF) of target set size per call site. The goal is to have as many call sites as possible and a very short tail, indicating few call sites with many targets, as such call sites are easily exploitable. For example, in case of 400.perlbench 5(a), most of the call sites have very few targets, 65% of all call sites have only one target. Similar situations were observed in the 445.gobmk benchmark; where the maximum target set size for LLVM-CFI is 1642, compared to 492 for Ancile. In all of these benchmarks, Ancile has fewer targets than LLVM-CFI as well as the maximum number of targets allowed by any call site is on average 59% smaller. Table 2

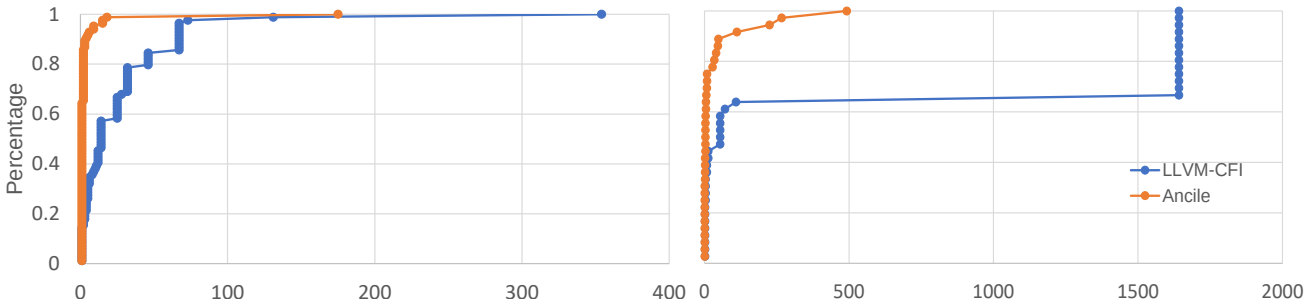
Table 2: Statistics of maximum target size in LLVM-CFI and Ancile for our benchmarks. An 'x' indicates a benchmark without any recorded indirect calls.

Benchmark	Max. target set size	
	LLVM-CFI	Ancile
400.perlbench	354	175
401.bzip2	1	1
403.gcc	366	38
429.mcf	x	x
433.milc	2	2
444.namd	40	1
445.gobmk	1642	492
447.dealII	11	2
450.soplex	7	1
453.povray	81	9
456.hmmer	11	1
458.sjeng	10	6
462.libquantum	x	x
464.h264ref	12	10
470.lbm	x	x
471.omnetpp	172	168
473.astar	1	1
482.sphinx3	5	1
483.xalanbmk	100	38
libtiff	78	16 (testsuite)
libpng	48	25 (testsuite)
nginx	103	87 (testsuite)
tcpdump	130	18 (testsuite)



(a) libtiff callsites (b) libpng callsites

Figure 4: Comparison of number of targets per each callsite at LLVM-CFI and Ancile with specialization in different functionalities for two libraries: libtiff and libpng. For each case study, we analyzed LLVM-CFI and Ancile in three different scenarios: standard test-suite along with two utilities (*tiffcrop* and *tiff2pdf* utilities for libtiff, and *pngfix* and *timepng* utilities for libpng).



(a) 400.perlbench targets (b) 445.gobmk targets

Figure 5: Comparison of the cumulative distribution function (CDF) of the target set size per call site of Ancile against LLVM-CFI over two SPEC CPU2006 benchmarks: 400.perlbench and 445.gobmk

shows the maximum target set size in LLVM-CFI and Ancile for each of the evaluated benchmarks.

Equivalence Classes. Equivalence classes are an important part of static analysis-based CFI. Each class is a group of call sites that are all assigned to the same target set (e.g., based on function prototypes). Ancile does away with the notion of equivalence classes as each call site is independently analyzed, instead of being grouped together as per existing static analysis-based approaches. In other words, Ancile introduces an equivalence class for each indirect call instead of, in its most precise form, for each function pointer type for LLVM-CFI. Having more equivalence classes increases the security of applications [17], as each call site has the minimum target set appropriate for it, not the target set for a class of call sites.

Ideally, we would increase the number of classes while reducing the size of each class. Ancile breaks large equivalence classes into smaller ones, namely one class per indirect call site, thus restricting the indirect calls to fewer targets. Ancile always generates more equivalence classes than LLVM-CFI, and the classes are strictly smaller, in most cases restricting the call site to single target.

6.3 RQ3: Correctness of specialized binaries

To confirm the correctness of Ancile-generated binaries, we performed a series of analyses such as result consistency, assessment of target discovery, correctness of generated input, target set minimality, and statistical analysis.

Consistency. One way to establish the confidence of the result is to check for consistency. If two separate fuzzer can generate same set of targets, it can increase our confidence in the specialized binary. We have used two separate fuzzers, AFL and honggfuzz, to generate the dynamic CFG and we achieved similar outcomes.

Target Discovery. Using fuzzing for target discovery comes with the challenge of effectiveness in learning targets. To understand this aspect, we plotted the discovery of each unique target against time. Figure 6 shows the number of targets discovered over time by the fuzzer for *tcpdump* with the command line option `r` for reading IPv4 and IPv6 captured packets. The `x`-axis

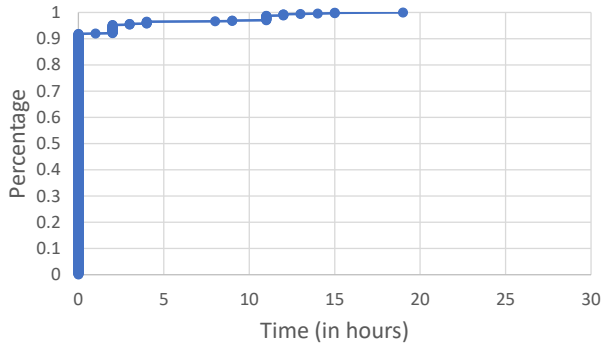


Figure 6: Target discovery over the time during application (tcpdump) fuzzing.

plots time in hour and y-axis plots the percentage of target discovery. From the figure, it is evident that most of the targets are discovered at the very beginning of the fuzzing procedure and few to no new targets towards later phases of fuzzing. This same observation holds true for all programs we tested. Furthermore, we reran all the fuzzing executions multiple times and target discovery remain identical in all the fuzzing sessions.

This profile of target discovery, with most targets discovered early, increases our confidence that fuzzing is finding all possible targets, and that continuing to fuzz for greater than 24 hours will not find additional targets.

Correctness of Generated Input. In order to cross-check that the fuzzer generated executions are valid, we applied several sanitizers (ASan, Ubsan) to check the correctness of fuzzer generated inputs. We also manually ensured that for each of these generated inputs there is an intended control-flow execution.

Minimality. Almost all dynamic CFI policies [38] have a fallback strategy and they usually fall back to over-approximated target sets generated statically. Ancile is inherently more aggressive. Although it uses instrumentation similar to LLVM-CFI for its enforcement, it never reduces precision to LLVM-CFI target sets. Ancile considers any call site or target that has not been exercised during profiling phase as invalid or, in other words, not relevant to the intended functionality. This is to ensure that we only employ the desired functionality. Our investigation indicates that this reduction has a *meaningful* impact on the application's security by making sensitive functions harder to access (more levels of indirection are required) from indirect call sites.

Statistical Analysis. A potential issue of using fuzzing is that the fuzzer may include superfluous coverage, i.e., the fuzzer discovers functionality that the user does not want included, preferably known as false negative. One way to handle this situation is to tune the length of the fuzzing campaigns. For example, when extracting functionality of reading the captured `pcap` packets using *tcpdump*, it is unlikely that the fuzzer will mutate the input seed enough to discover the code that handles capturing packets. Due to the stochastic nature of fuzzing, it is also possible that Ancile might miss some intended control flows resulting in false positives.

Table 3: Performance overhead comparison between LLVM-CFI and Ancile. Note that LLVM-CFI could not compile—and run—some benchmarks (marked by '-').

Benchmark	Baseline (ms)	LLVM-CFI (ms)	Ancile (ms)
400.perlbench	374	379 (1.33%)	378 (1.07%)
401.bzip2	726	730 (0.55%)	730 (0.55%)
403.gcc	781	-	790 (1.1%)
429.mcf	296	297 (0.34%)	297 (0.34%)
433.milc	1029	1037 (0.78%)	1036 (0.68%)
444.namd	1420	1429 (0.63%)	1430 (0.70%)
445.gobmk	518	522 (0.77%)	519 (0.19%)
447.dealII	1294	1301 (0.54%)	1300 (0.46%)
450.soplex	339	345 (1.78%)	345 (1.78%)
453.povray	440	-	451 (2.5%)
456.hmmer	569	-	572 (0.52%)
458.sjeng	620	621 (0.16%)	622 (0.32%)
462.libquantum	474	481 (2.34%)	481 (2.34%)
464.h264ref	872	877 (0.57%)	879 (0.80%)
470.lbm	692	695 (0.43%)	694 (0.28%)
471.omnetpp	781	-	802 (2.6%)
473.astar	544	546 (0.33%)	546(0.33%)
482.sphinx3	945	947 (0.21%)	946 (0.11%)
483.xalanbmk	1325	-	1341(1.2%)

To understand how Ancile performs with respect to false positives and false negatives, we have analyzed it with forty different test cases for each of our case studies. In half of our test cases, we analyzed the specialized binary with the same intended functionality but with different set of inputs. For example, in case of *tiff2pdf* utility, we evaluated it with twenty different *tif* files which we have not used as seed. In similar way, we have used the rest twenty of the test cases to exercise an unintended functionality. Ancile successfully validated all test scenarios for all the investigated applications.

In future work, we will evaluate how a user can select negative functionality they want explicitly excluded. We refer to existing work that focused on similar challenges [37].

6.4 RQ4: Performance Overhead

Performance overhead is crucial for mitigations. We analyzed the performance of Ancile on SPEC CPU2006 and compared it with LLVM-CFI. Table 3 presents a comparison of runtime performance of Ancile and LLVM-CFI. Ancile's enforcement mechanism mainly reuses the enforcement part of LLVM-CFI with a tighter target set, and as the table shows, has equivalent runtime performance. As is standard, we report results for three SPEC CPU2006 iterations. Note that we require no additional system resources, such as additional processes, cores, virtual address space, or hardware extensions, unlike other works aimed at increasing the precision of CFI [21, 27, 53].

7 RELATED WORK

Software Debloating is a well-known attack mitigation scheme which reduces code size and complexity. Rastogi et al. introduced *Cimplifier* [43], an approach for debloating containers by using dynamic analysis for necessary resource identification. *Chisel* [29] debloats programs at a fine-grained

level through reinforcement learning. *Trimmer* [49] eliminates unused functionality based on user-provided configuration data. Quanch et al. [41] debloat programs via piece wise compilation and loading. They analyze the program to build a dependency graph of external functions and then only load the required functions as well as remove any library code. Nibbler [14] performs similar library specialization at the binary level. BinTrimmer [45] utilizes abstract interpretation to recover a precise CFG as well as to identify unreachable code and then removing it. Unfortunately, software debloating is not enough to stop CFH. An attacker can still exploit bugs in the remaining code segments and launch code-reuse attacks.

Razor [40] is another post-deployment debloating framework which works at the binary level. It has three components: Tracer, Path finder and Generator. It debloats the binary by utilizing test cases to trace execution paths, then uses four heuristics to find nearby code-paths. Finally, the generator rewrites the binary. Similar to Razor, Binary Control-Flow Trimming[28] uses test traces and later machine learning to explore relevant control-flows. Both of these works are binary based and utilize test traces, where as Ancile works primarily on source code and it depends on the user given seeds to map functionality into code. The main distinction of Ancile over these two works is it introduces seed demonstrated fuzzing to explore relevant code regions. It strengthens the security of an application by not only debloating unused functionalities, but also eliminating invalid targets from the remaining control transfers.

Control-Flow Integrity reduces attack surface by prohibiting illegal control flow transfers from the CFG. After the introduction of the CFI mechanism by Abadi et al. [13] in 2005, the mechanism saw a diverse set of improvements along performance, security, and precision. For a full survey see Burow et. al [17].

LLVM-CFI [5] is a static analysis based CFI approach that is implemented in production compilers with negligible overhead (approximately 1%) [4]. In this approach, each indirect call along with associated targets are clustered into equivalence classes where each indirect call can target any of the addresses within the associated equivalence class. However, due to the reliance on the static analysis, LLVM-CFI struggles with aliasing that results in an over-approximation. An attacker can perform attacks [19, 24, 33, 47] by leading an indirect control flow to a different target within the equivalence class without violating the CFG. LLVM-CFI is seeing wide deployment by Google in Chrome [20] and Android [15].

Recent research efforts improved the precision, and thus security, of CFI. *PittyPat* [21] presents a path-sensitive approach combining hardware-based monitoring and runtime points-to-analysis. It improves preciseness with the cost of additional hardware and performance overhead. In particular, it requires a separate process to monitor and validate the execution traces of the protected process. πCFI [38] starts enforcement of a process with an empty CFG and adds edges dynamically by activating addresses as needed. The security of πCFI depends on an attacker's inability to activate certain edges, otherwise it would provide the same guarantees as a static CFI policy (modulo the complexity of activating the target). VIP [25] adds a measure of control and data-flow sensitivity to the static analysis used by CFI. Ancile achieves greater precision than πCFI or VIP through its functional analysis, and does not require additional system resources like PittyPat.

Existing solutions for control-flow hijacking cannot protect against data-flow attacks and leave the attacker some room. Ancile restricts the application to the bare minimum code required to run the specified functionality and thereby restricts the power of data-only attacks to this exposed functionality. If there is no path to, e.g., `execve` then no modification of the program's memory can bend the control flow to the sensitive function.

8 CONCLUSION

We present Ancile, a code specialization technique through fuzzing. Our case studies show that seed demonstrated fuzzing can be used to effectively map user-intended functionalities into relevant code regions. We can then leverage this information to guide debloating and program specialization, reducing the program's attack surface and improving the precision of defenses such as CFI.

We believe that automatically specializing code for particular usage scenarios via fuzzing is a promising new technique for software security. It can achieve greater security than static analysis without requiring extra system resources.

9 ACKNOWLEDGMENTS

We thank the reviewers and our shepherd Roland Yap for their feedback. This project has received funding from the European Research Council (ERC) under grant agreement No. 850868, NSF under grant CNS-1801601, and a gift from Huawei. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Online; accssed 10-Oct-2020. Fuzzing. https://www.owasp.org/index.php/Fuzzing.
[2] Online; accssed 11-Oct-2020. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[3] Online; accssed 11-Oct-2020. Control-flow Enforcement Technology. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.
[4] Online; accssed 11-Oct-2020. Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html.
[5] Online; accssed 11-Oct-2020. Control Flow Integrity Design Documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.
[6] Online; accssed 11-Oct-2020. honggfuzz. https://github.com/google/honggfuzz.
[7] Online; accssed 11-Oct-2020. libpng. http://www.libpng.org/pub/png/libpng.html.
[8] Online; accssed 11-Oct-2020. libTIFF. http://www.libtiff.org/.
[9] Online; accssed 11-Oct-2020. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[10] Online; accssed 13-Oct-2020. Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/.
[11] Online; accssed 13-Oct-2020. Linux Kernel. https://en.wikipedia.org/wiki/Linux_kernel.
[12] Online; accssed 15-Jan-2021. CVE-2014-0038: Privilege escalation in X32 ABI. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038.

[13] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[14] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 70–83.

[15] Google Android. 2018. Kernel Control Flow Integrity. https://source.android.com/devices/tech/debug/kcfi.

[16] Michael D Brown and Santosh Pande. 2019. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19)*.

[17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys(CSUR)* 50, 1 (2017), 16.

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. *SP'19* (2019).

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity.. In *USENIX Security Symposium*. 161–176.

[20] Google Chromium. 2017. Chromium: Control Flow Integrity. https://www.chromium.org/developers/testing/control-flow-integrity.

[21] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 131–148.

[22] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction, CC*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). 132–142.

[23] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society.

[24] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 901–913.

[25] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[26] Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 396–405.

[27] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 585–598.

[28] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1009–1022.

[29] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 380–394.

[30] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. 2012. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *WOOT*.

[31] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1470–1486.

[32] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 969–986.

[33] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1868–1882.

[34] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. 2020. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *Usenix Annual Technical Conference*.

[35] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis. In *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE'18)*.

[36] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[37] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. 2014. Reuse-oriented Reverse Engineering of Functional Components from x86 Binaries. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1128–1139. https://doi.org/10.1145/2568225.2568296

[38] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 914–926.

[39] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[40] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A Framework for Post-deployment Software Debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1733–1750.

[41] Anh Quach, Aravind Prakash, and Lok Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. *arXiv preprint arXiv:1802.00759* (2018).

[42] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[43] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 476–486.

[44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[45] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. B in T rimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 482–501.

[46] Address Sanitizer. Online; accssed 10-Oct-2020. *ASan*.

[47] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 745–762.

[48] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[49] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 329–339.

[50] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2018. SoK: Sanitizing for Security. *arXiv preprint arXiv:1806.04355* (2018).

[51] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 46–55.

[52] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.. In *USENIX Security Symposium*. 941–955.

[53] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 927–940.