Lightweight, Multi-Stage, Compiler-Assisted Application Specialization

Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps
University of Wisconsin-Madison, USA
{Mohannad,Rithik,Jha,Reps}@cs.wisc.edu, vaibhavrastogi@google.com

2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1614-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EUROSP53844.2022.00024

***Abstract*—Program debloating aims to enhance the performance and reduce the attack surface of bloated applications. Several techniques have been recently proposed to specialize programs. These approaches are either based on unsound strategies or demanding techniques, leading to unsafe results or a high-overhead debloating process. In this paper, we address these limitations by applying partial-evaluation principles to generate specialized applications. Our approach relies on a simple observation that an application typically consists of configuration logic, followed by the main logic of the program. The configuration logic specifies what functionality in the main logic should be executed. LMCAS performs partial interpretation to capture a precise program state of the configuration logic based on the supplied inputs. LMCAS then applies partial-evaluation optimizations to generate a specialized program by propagating the constants in the captured partial state, eliminating unwanted code, and preserving the desired functionalities.**

Our evaluation of LMCAS—on commonly used benchmarks and real-world applications—shows that it successfully *removes unwanted features* while *preserving the functionality and robustness* of the debloated programs, *runs faster than prior tools*, and *reduces the attack surface* of specialized programs. LMCAS runs 1500x, 4.6x, and 1.2x faster than the state-of-the-art debloating tools CHISEL, RAZOR, and OCCAM, respectively; achieves 25% reduction in the binary size; demonstrates favorable gadgets elimination trade-off; and eliminates 87.5% of the known CVE vulnerabilities in our test corpus.

***Index Terms*—Debloating, Specialization, Security, Program Division, Partial Evaluation**

1. Introduction

The software stack is becoming increasingly bloated. This software growth decreases performance and increases security vulnerabilities. *Software debloating* is a mitigation approach that downsizes programs while retaining certain desired functionality. Although static program debloating can thwart unknown possibilities for attack by reducing the attack surface [38], prior work has generally not been effective, due to the overapproximation of static program analysis: these tools determine statically the set of functions to be removed, using a combination of analysis techniques, such as unreachable-function analysis and global constant propagation [45], but a lot of bloated code remains. More aggressive debloating approaches (e.g., RAZOR [46] and Chisel [25]) can achieve more reduction; however, they involve demanding techniques: the user needs to define a comprehensive set of test cases to cover the desired functionalities, generate many traces of the program, and perform extensive instrumentation. The computational expense of these steps leads to a high-overhead debloating process. The work on RAZOR acknowledges the challenge of generating test cases to cover all of the desired code, and incorporates heuristics to address this challenge. Furthermore, these aggressive approaches often break soundness, which can lead the debloated programs to crash or execute incorrectly. These issues make such approaches unsafe and impractical [45].

Partial evaluation is a promising program-specialization technique. It is applied in prior work [37], [53], but prior work has suffered from the overapproximation inherent in static program analysis. In particular, existing implementations rely only on the command-line arguments to drive the propagation of constants. However, they do not capture precisely the set of variables that are affected by the supplied inputs. Constant propagation is performed only for global variables that have one of the base types (`int`, `char`), and no attempt is made to handle compound datatypes (`struct`). Such a limited approach leaves a substantial amount of unwanted code in the "debloated program," which reduces the security benefits.

In this paper, we present Lightweight Multi-Stage Compiler-Assisted Application Specialization (LMCAS), a new software-debloating framework. LMCAS relies on the observation that, in general, programs consist of two components: (a) *configuration logic*, in which the inputs are parsed, and (b) *main logic*, which implements the set of functionalities provided by the program. We call the boundary between the two components the *neck*. LMCAS captures a partial state of the program by interpreting the configuration logic based on the supplied inputs. The partial state comprises concrete values of the variables that are influenced by the supplied inputs, LMCAS then applies partial-evaluation optimizations to generate the specialized program. These optimizations involve converting the influenced variables at the neck into constants, applying constant propagation, and performing multiple stages of standard and custom compiler optimizations.

LMCAS makes significant and novel extensions to make debloating much safer in a modern context. The extensions involve optimizing the debloating process and improving its soundness. Specifically, we optimize the debloating process by introducing the *neck* concept, which eliminates demanding techniques that require: (1) executing the whole program, (2) performing extensive instrumentation, and (3) obtaining a large set of tests. We

DOI 10.1109/EuroSP53844.2022.00024

demonstrate the soundness of our approach by validating the functionality of 25 programs after debloating, under various settings. The achieved soundness is driven by: capturing a precise partial state of the program, supporting various data types, and performing guided constant conversion and clean-up.

Our evaluation demonstrates that LMCAS is quite effective: on average, LMCAS achieves 25% reduction in the binary size with moderate overall ROP-gadget removal and minimal rates of ROP gadget introduction in the specialized applications. LMCAS eliminates 87.5% of known CVE vulnerabilities. On average, LMCAS runs 1500x, 4.6x, and 1.2x faster than the state-of-the-art debloating tools CHISEL, RAZOR, and OCCAM, respectively. Hence, LMCAS strikes a favorable trade-off between functionality, performance, and security.

The contributions of our work are as follows:

1) We propose a novel debloating idea that takes advantage of the common pattern that a program has a natural division into configuration logic and main logic to reduce the overhead of debloating process.
2) We develop a neck miner to enable the identification of such a boundary with high accuracy, substantially alleviating the amount of manual effort required to identify the neck.
3) We apply the principles of partial evaluation to generate specialized programs based on supplied inputs. A partial program state is captured by conducting partial interpretation for the program by executing the configuration logic according to the supplied inputs. The program state is enforced by applying compiler optimizations for the main logic to generate the specialized program.
4) We develop the LMCAS prototype based on LLVM. LMCAS harnesses symbolic execution to perform partial interpretation and applies a set of LLVM passes to perform the compiler optimizations. We also carried out an extensive evaluation based on real-world applications to demonstrate the low overhead, size reduction, security, and practicality of LMCAS.
5) We will make LMCAS implementation [1] and its artifacts available to the community.

2. Motivation and Background

This section presents a motivating example (§2.1), and reviews necessary background material on program analysis and software debloating used in the remainder of the paper (§2.2). We discuss debloating challenges illustrated by the motivating example, and describe our solutions for addressing these challenges (§2.3). Finally, we discuss the threat model that our work addresses (§2.4).

2.1. Motivating Example

Listing 1 presents a scaled-down version of the UNIX word-count utility `wc`. It reads a line from the specified stream (i.e., `stdin`), counts the number of lines and/or characters in the processed stream, and prints the results. Thus, this program implements the same action that would be obtained by invoking the UNIX word-count utility with the `-lc` flag (i.e., `wc -lc`). Although Listing 1 merely supports two counting options, it is still bloated if the user is only interested in enabling the functionality that counts the number of lines.

```
1  struct Flags {
2    char count_chars;
3    int count_lines; }
4  int total_lines = 0;
5  int total_chars = 0;
6  int main(int argc, char** argv){
7    struct Flags *flag;
8    flag = malloc(sizeof(struct Flags));
9    flag->count_chars = 0;
10   flag->count_lines = 0;
11   if (argc >= 2){
12     for (int i = 1; i < argc; i++) {
13       if (!strcmp(argv[i], "-c")) flag->count_chars = 1;
14
         if (!strcmp(argv[i], "-l")) flag->count_lines = 1; }}
15   char buffer[1024];
16   while (fgets(buffer, 1024,stdin)){
17
         if (flag->count_chars) total_chars += decodeChar(buffer);
18     if (flag->count_lines) total_lines++;}
19
         if (flag->count_chars) printf("#Chars = %d", total_chars);
20   if (flag->count_lines) printf("#Lines = %d", total_lines
       ); }
```

Listing 1. A scaled-down version of the `wc` utility. Highlighted statements are eliminated after debloating with "`wc -l`".

Put another way, Listing 1 goes against the *principle of least privilege* [31] [50]: code that implements unnecessary functionality can contain security vulnerabilities that an attacker can use to obtain control or deny service—bloated code may represent an opportunity for privilege escalation. For instance, the character-count functionality "`wc -c`" processes and decodes the provided stream of characters via the function `decodeChar` (Line 17 of Listing 1). An attacker might force it to process special characters that `decodeChar` cannot handle [57]. More broadly, attackers can supply malicious code as specially crafted text that injects shellcode and thereby bypasses input restrictions [39]. Debloating is a way to reduce a program's attack surface: the "`wc -l`" functionality does not require the character-processing code used in "`wc -c`", and the call to the function `decodeChar` is completely absent in the specialized version for "`wc -l`".

2.2. Background

Partial evaluation [27], [51], [17] is an optimization and specialization technique that precomputes program expressions in terms of the known static input—i.e., a subset of the input that is supplied ahead of the remainder of the input. The result is another program, known as the *residual program*, which is a specialization of the original program. To create the residual program, a partial evaluator performs optimizations such as loop unrolling, constant propagation and folding, function inlining, etc. [27].

Partial evaluation and symbolic execution [28] both generalize standard interpretation of programs, but there are key differences between them. Specifically, symbolic

1. https://github.com/Mohannadcse/LMCAS_Demo

execution interprets a program using symbolic input values, while partial evaluation precomputes program expressions and simplifies code based on the supplied inputs. Unlike partial evaluation, one result of symbolic execution is a set of expressions for the program's output variables. Because their capabilities are complementary, the two techniques have been employed together to improve the capabilities of a software-verification tool [14].

LLVM [34] provides robust compiler infrastructures for popular programming languages, including C and C++, and supports a wealth of compiler analyses and optimizations that make it suitable for developing new compiler transforms. LLVM operates on its own low-level code representation known as the LLVM intermediate representation (LLVM IR). LLVM is widely used in both academia and industry.

2.3. Challenges and Solutions

In this section, we formalize the program-specialization problem illustrated in Listing 1. In general, there is (i) a program P that provides a set of functionalities F, and (ii) an input space I that contains a set of values that enable certain functionalities in F. Typically, one or more functionalities are enabled based on a set of supplied inputs I_s, which are provided as part of a command-line argument or configuration file. Generating a specialized program P' based on the set of supplied inputs I_s requires identifying a set of variables $V_s = \{v_0, v_1, .., v_n\}$ that are influenced by the supplied inputs, and a corresponding set of constant values $C_s = \{c_0, c_1, .., c_n\}$. The relationship between V_s and C_s is bijective and $I_s \subset I$. To generate a specialized program P' that (i) retains the required functionalities based on the supplied inputs I_s, and (ii) removes irrelevant functionalities, we need to address the challenges discussed below:

Challenge 1: how to *optimize* the debloating process and avoid high-demand techniques?

Solution. To address this challenge, we propose to interpret the program partially up to a certain point, instead of executing the whole program. We can achieve this partial interpretation by relying on the observation that, programs often consist of two components: (a) *configuration logic*, in which the inputs (from the input space I) are parsed, and (b) *main logic*, which implements the set of functionalities F. We call the boundary point the *neck*. The frequent occurrence of a configuration-logic-to-main-logic transition has been exploited by others. For instance, Ghavamnia et al. [24] observed that in server programs there is often such a boundary between initialization and serving phases, which they took advantage of to perform temporal debloating. (They also observed that the concept often applies to client programs.) Meinicke et al. [40] studied some of the reasons why programmers introduce and respect such a separation, and proposed several recommendations to regularize the practice and to document the configuration parameters and features that are supported in a program by this means.

The *partial interpreter* needs only part of the program state to be available by executing the program up to the neck. By this means, we obtain a precise characterization of the set of variables V_s that are influenced by the supplied arguments. We then convert these variables to constants, based on the constant values C_s identified by the partial interpreter. These values are then propagated to other parts of the program via partial evaluation.

Consider again Listing 1. The program `wc` provides two functionalities: $F = \{$*counting_lines*, *counting_characters*$\}$, and these functionalities can be activated through the two inputs $I = \{l, c\}$. For generating the specialized program P' that retains the *counting_lines* functionality (i.e., "`wc -l`") based on the supplied input $I_s = \{l\}$, we interpret program P up to the neck (i.e., Line 15) to identify the set of influenced variables $V_s = \{$`flag->count_chars`, `flag->count_lines`, `total_lines`, and `total_chars`$\}$, together with the corresponding constant values $C_s = \{0, 1, 0, 0\}$ (the partial state of P). We supply this information to the partial evaluator to generate the specialized program.

Challenge 2: how to simplify the program *sufficiently*, while ensuring that it operates *correctly*, and preserve its functionality and soundness?

Solution. The combination of *partial interpretation* followed by partial-evaluation optimizations holds the promise of achieving significant debloating. To achieve this promise and preserve the program semantics, it is necessary to handle various data types and complex data structures (i.e., strings, pointers, and structs). By using a precise model of the programming language's semantics, more information about variables and their values is made available, which in turn enables more optimizations to be carried out during program specialization. Therefore, we need to capture a broad spectrum of variables. For instance, the scaled-down word-count in Listing 1 contains a stack variable (`flag`) and two global variables (`total_lines` and `total_chars`). Various data types need to be supported as well: the global variables are integers, whereas the stack variable `flag` is a pointer to a struct that consists of two fields (`count_line` and `count_chars`). Supporting these various types of variables provides LMCAS the capability to perform safe debloating and maintain soundness.

2.4. Threat Model

From a security standpoint, the basic premise of debloating is that reducing code size decreases a program's attack surface. In this work, we consider remote adversaries that exploit an existing vulnerability that allows arbitrary code execution. The attack vector for these vulnerabilities can involve local interactions (i.e., by an on-premise user) or interactions over the network. The classification of these attack vectors is according to the Common Vulnerability Scoring System (CVSS) [20], an industry framework for communicating the characteristics and severity of software vulnerabilities. Our debloating technique aims to limit the attacker's capabilities to run an exploit; the debloater can (i) manipulate the control flow in the debloated program, and (ii) remove code at various levels of granularity (i.e., functions, basic blocks, and LLVM IR instructions) according to the supplied inputs. Code removal should eliminate known vulnerabilities related to disabled features in the specialized

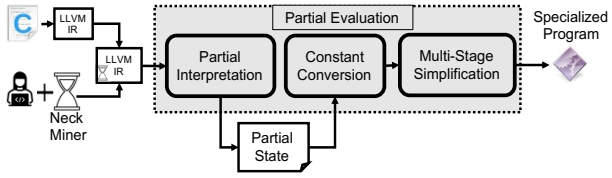

Figure 1. LMCAS Workflow.

programs. For instance, in Listing 1, the exploitable function `decodeChar` should be completely absent in the specialized version for "`wc -l`". The manipulation of control flow by the debloater can alter the semantics of any gadgets for return-oriented programming and allied techniques [47], [9] that are present in executable memory when the program runs, without affecting the intended semantics of the actual program code [43].

3. LMCAS Framework

This section introduces LMCAS, a lightweight debloating framework that uses sound analysis techniques to generate specialized programs. Figure 1 illustrates the architecture of LMCAS. The debloating pipeline of LMCAS receives as input the whole-program LLVM bitcode of program P, and performs the following major phases to generate a specialized program P' as bitcode, which is ultimately converted into a binary executable.

- **Neck Miner (§3.1).** Receives P, and modifies it by adding a special call that marks the neck. §3.1 describes our approach to neck identification based on heuristic and structural analysis.
- **Partial Interpretation (§3.2).** Interprets the program up to the neck (i.e., terminates after executing the special function call inserted by the neck miner), based on the supplied inputs that control which functionality should be supported by the specialized application. The output of this phase provides a precise partial state of the program at the neck. This partial state comprises the variables that have been initialized during the partial interpretation and their corresponding values.
- **Constant Conversion (§3.3).** Incorporates into the program the partial state captured by the partial interpretation. It converts the variables and their corresponding values captured in the partial state (i.e., V_s and C_s) to settings of constants at the neck. This phase also provides the opportunity to boost the degree of subsequent optimization steps by supporting the conversion of multiple kinds of variables to constants.
- **Multi-Stage Simplification (§3.4).** Applies selected standard and customized LLVM passes for optimizing the program and removing unnecessary functionality. These optimization steps are arranged and tailored to take advantage of the values introduced by the constant-conversion phase.

3.1. Neck Miner

We developed a neck miner to recommend potential neck locations. To illustrate the neck idea, consider the example in Figure 2, which represents the motivating example of Listing 1, but with the split between configuration

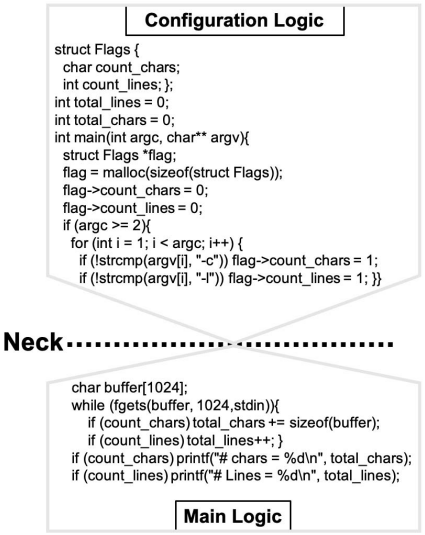

Figure 2. The neck miner selects Line 15 as a splitting point to partition the motivating example in Listing 1 into *configuration logic* and *main logic*. The splitting point is called the neck.

logic and main logic emphasized. The configuration-logic part consists of Lines 4-14, which include the declaration and initialization of global variables `total_lines` and `total_chars`, and the first part of function `main`. The rest of `main` (Lines 15-20) represents the main logic, which contains the functionalities of counting lines and counting characters. We call the point at the boundary between the two components the *neck*. Because the neck is located at the end of the configuration logic—where core arguments are parsed—the neck location is independent of the values of supplied inputs. Thus, neck identification only needs to be conducted a single time for each program, and the neck location can be reused for different inputs—i.e., for different invocations of the debloater on that program. According to the motivating example (Listing1), the same neck location at Line 15 can be used for debloating whether the supplied input is "`wc -l`" or "`wc -c`" .

The neck miner uses two analyses: *heuristic analysis* and *structural analysis*, as described in Algorithm 1. The *heuristic analysis* relies on various patterns, corresponding to *command-line* and *configuration-file* programs, to identify a location from which to start the *structural analysis*, which identifies the neck properly.

3.1.1. Heuristic Analysis. This step guides the structural analysis. It identifies a single location from which the structural analysis can be conducted, and relies on a set of patterns that apply to two categories of programs. These patterns are described as follows:

Command-Line-Program Patterns (Algorithm 1 Lines 5–9): the inputs are provided to this category of programs via command-line arguments. In C/C++ programs, command-line arguments are passed to `main()` via the parameters `argc` and `argv`: `argc` holds the number of command-line arguments, and `argv[]` is a pointer array whose elements point to the different arguments passed to the program. Consequently, this analysis step tracks the use of the argument `argv` (Line 6). Specifically, because `argv` is a pointer array that, in general, points to multiple arguments, the analysis identifies the uses of `argv` that

Algorithm 1: Neck Miner Algorithm

```
   Input: CFG, EntryPoint, programCategory, fileParsingAPIs
   Output: NeckLocation
1  startingPointForStructuralAnalysis = NULL
2  distanceToNeckLoc ← ∅
3  distanceToInst ← ∅
4  /* Heuristic Analysis                                   */
5  if programCategory is Command-Line then
6      for inst ∈ UsesOf(argv) do
7          if inst is not inside a loop-structure then
8              distance = computeDistance(inst,EntryPoint,CFG)
9              distanceToInst ← distanceToInst ∪ pair(inst, distance)
10 else if programCategory is Config-File then
11     for inst ∈ CFG do
12         if inst ∈ fileParsingAPIs then
13             distance = computeDistance(inst,EntryPoint,CFG)
14             distanceToInst ← distanceToInst ∪ pair(inst, distance)
15 startingPointForStructuralAnalysis =
     InstAtShortestDistance(distanceToInst)
16 /* Structural Analysis                                  */
17 for inst ∈ CFG do
18     if inst is after startingPointForStructuralAnalysis in CFG then
19         if inst satisfies the control-flow properties from §3.1.2 then
20             distance = computeDistance(inst,EntryPoint,CFG)
21             distanceToNeckLoc ←
                 distanceToNeckLoc ∪ pair(inst, distance)
22 NeckLocation = InstAtShortestDistance(distanceToNeckLoc)
23 Add special function call before NeckLocation to mark the neck
```

are inside of a loop (Line 7).

Configuration-File-Program Patterns (Algorithm 1 Lines 10–14): this category of programs relies on configuration files to identify the required functionalities. For instance, consider how the neck is identified in Nginx, a web-server program that supports 724 different configuration options [29]. Listing 2 presents a simple Nginx configuration file. The gzip directive at line 7 is associated with the libz.so library. In some cases, multiple directives have to be defined to enable a certain capability, such as the SSL-related directives in lines 6, 8, and 9 of Listing 2. The heuristic analysis specifies the first location where the configuration file is parsed by certain APIs. Identifying such APIs is simple because programs typically use system calls to read files. For instance, nginx uses the Linux system call `pread`[2] to read the configuration file. If the program uses custom parsing functions, either located in the codebase, or in a library, then the LMCAS user needs to supply the names of those functions to the heuristic analysis. If the functions are in a dynamically linked library, they can be made visible to LMCAS by statically linking the library.

```
1  worker_processes 1;
2  http {
3    charset UTF_8;
4    keepalive_timeout 65;
5    server {
6      listen 443 ssl;                  # libssl.so
7      gzip on;                         # libz.so
8      ssl_certificate cert.pem;        # libssl.so
9      ssl_certificate_key cert.key;    # libssl.so
10   } }
```

Listing 2. Nginx configuration file

Finally, after identifying the set of statements that match the various patterns, the heuristic analysis returns the statement that is closest to the CFG's entry point (Line 15), and ties are broken arbitrarily. In the motivating example (Listing 1), the heuristic analysis obtains the statement at Line 13 because it is the closest location to the entry point that matches the command-line patterns.

2. https://man7.org/linux/man-pages/man2/pwrite.2.html

3.1.2. Structural Analysis. This step identifies the neck location by analyzing the program's statements, starting from the location specified by the heuristic analysis (Lines 17- 18 in Algorithm 1). It identifies the statements that satisfy a certain set of control-flow properties that are discussed below (Line 19). Because it is possible to have several matching statements, the closest statement to the entry point is selected (Line 22). (Ties are broken arbitrarily.) The closest statement is determined by computing the shortest distances in the CFG from the entry point to the neck candidates (Line 20). The remainder of this section formalizes the aforementioned control-flow properties.

A program P is a 4-tuple $(V, stmts, entry, exit)$, where V is the set of variables, $stmts$ is the set of statements, $entry \in stmts$ is the entry point of the program, and $exit \in stmts$ is the exit of the program. As defined in Section 2.3, we assume that there is a set $V_s \subseteq V$, which we call the set of *influenced variables* (e.g., command-line parameters of a utility). Note that $V - V_s$ is the set of "internal" or non-influenced variables. The location of a statement $s \in stmt$ is denoted by $loc(s)$. For simplicity, we assume that Val is the set of values that vars in V can take.

Let $A : V_s \rightarrow Val \cup \{\star\}$ be a partial assignment to the set of influenced variables (we assume that if $A(v) = \star$, then it has not been assigned a value). An assignment $A' : V_s \rightarrow Val$ is *consistent* with partial assignment A iff for all $v \in V_s$, if $A(v) \neq \star$, then $A'(v) = A(v)$. A statement $s \in stmt$ is a *neck* for a program $P = (V, stmts, entry, exit)$ and a partial assignment A (denoted by $nk(P, A)$) if the following conditions hold:

- Given any assignment A' consistent with A, P always reaches the neck $nk(P, A)$, and the statement corresponding to **the neck is executed exactly once**. This condition rules out the following possibilities: Given A', the execution of P (i) might never reach the neck (the intuition here is that we do not want to miss some program statements, which would cause debloating to be unsound), or (ii) the statement corresponding to the neck is inside a loop.
- Let $R(nk(P, A))$ be all statements defined as follows: $s \in R(nk(P, A))$ iff s appears after $nk(P, A)$, then $nk(P, A)$ could be **identified as articulation point** of the CFG of P and one of the connected components would over-approximate $R(nk(P, A))$. Another structural condition could be defined as follows: $R(nk(P, A))$ is **the set of all statements that are dominated by the neck**.

In summary, the structural properties of the neck are: (i) it should dominate the main logic, and (ii) not be inside any loop structure (the neck is executed only once).

The neck miner is fully automated, except the step at Line 19 (in Algorithm 1), which currently requires manual intervention. We argue that such effort is manageable; moreover, it just requires a one-time effort for each program. This approach takes advantage of the tendency of developers to create disciplined and modular implementations that have a clean separation between configuration actions and the main computation of the program [40].

Once the developer identifies the statements that satisfy the control-flow properties, they are fed to the neck miner to identify the statement that is closest to the entry point in the CFG. Finally, a special function call (which serves to label the neck location) is inserted before the identified neck location.

Consider Listing 1 again. The developer iterates over the program code, starting from Line 13 (specified by the heuristic analysis) to identify the locations that satisfy the control-flow properties. The developer ignores Lines 13 and 14 because they violate the control-flow properties: they are not articulation points and are inside a loop, so not executed only once. Line 15 satisfies the control-flow properties because the statement at this location is executed only once, is an articulation point, and dominates all subsequent statements.

3.2. Partial Interpretation (PI)

Partial interpretation is a supporting phase whose goal is to identify—at the neck—the set of variables (and their values) that contribute to the desired functionality. Partial interpretation is performed by running a symbolic-execution engine (based on concrete inputs), starting at program entry, and executing the program (using partial program states, starting with the supplied concrete values) along a single path to the neck. For configuration-file programs, PI does not depend on the type of the configuration file (JSON, text, etc.). In principle, as long as it is feasible to partially evaluate the parsing function, the specified configuration can be captured regardless of the format of the configuration file. After partial interpretation terminates, the partial state is saved, and the values of all variables are extracted. Different types of variables are extracted, including base types (e.g., `int`, `char`) and constructed/compound types (e.g., `enum`, pointer, array, and `struct`).

Consider a network-monitoring tool, such as `tcpdump`, that supports multiple network interfaces, and takes as input a parameter that specifies which network interface to use. A specialization scenario might require monitoring a specific interface (e.g., Ethernet) and capturing a specific number of packets: for `tcpdump`, the command would be "`tcpdump -i ens160 -c 100`" (see the inputs listed in Table 8, Section 5). The first argument identifies the Ethernet interface `ens160`; the second argument specifies that 100 packets should be captured. The former argument is a string (treated as an array of characters in C); the latter is an `int`.

Returning to the example from §2 (Listing 1), Figure 2 illustrates the location of the neck. Suppose that the desired functionality is to count the number of lines (i.e., `wc -l`). Table 1 shows a subset of the variables and their corresponding values that will be captured and stored in LMCAS's database after partial interpretation finishes.

TABLE 1. PARTIAL PROGRAM STATE THAT CONTAINS THE SET OF CAPTURED VARIABLES V_s AND THE CORRESPONDING VALUES OBTAINED AT THE NECK AFTER PARTIAL INTERPRETATION OF LISTING 1.

Variable	Type	Scope	Value
`total_lines`	`int`	Global	0
`total_chars`			0
`flag->count_lines`	`int`	Local	1
`flag->count_chars`	`char`		0

3.3. Constant Conversion (CC)

This phase aims to propagate constants in the configuration logic to enable further optimizations. For instance, this phase contributes to removing input arguments that were not enabled during partial interpretation, and thus allows tests that check those inputs to be eliminated.

Constant conversion is a non-standard optimizing transformation because optimization is performed *upstream* of the neck: uses of variables in the configuration logic—which comes *before* the neck—are converted to constants, based on values captured at the neck. It also includes inter-procedural constant conversion that facilitates dead-code elimination and enforces accurate behaviour. This step is necessary to handle situations where some of the variables that are influenced by the supplied inputs are initialized before the neck. The transformations performed during this phase enforce that the state at the neck in the debloated program is consistent with the partial state of constants at the neck that was captured at the end of partial interpretation. Standard dataflow analyses (e.g. Def-Use) [22] for global and stack variables is used to replace all occurrences of the variables with their corresponding constant values in the program code before the neck. Because some of the program statements become dead after constant conversion, the replacement is performed for all occurrences (i.e., accesses) of the variables obtained after partial interpretation.

The CC phase receives as input the bitcode of the whole program P generated using WLLVM,[3] as well as a dictionary (similar to Table 1) that maps the set of variables in V_s captured after partial interpretation to their constant values C_s. The set V_s involves global and stack variables (base-type, struct, and pointer variables). The CC phase then iterates over the IR instructions to identify the locations where the variables are accessed, which is indicated by load instructions. Then, it replaces the loaded value with the corresponding constant value. This approach works for global variables and stack variables with base types. However, for pointers to base variables, it is necessary to identify locations where the pointer is modifying a base variable (by looking for store instructions whose destination-operand type is a pointer to a base type). The source operand of the store operation is modified to use the constant value corresponding to the actual base variable pointed to by the pointer.

For stack variables that are Structs and pointers to Structs, we first need to identify the memory address that is pointed to by these variables, which facilitates tracing back to finding the corresponding struct and pointer-to-struct variables. We then iterate over the use of the identified memory addresses to determine store operations that modify the variable (corresponding to the memory addresses). Finally, we convert the source operand of the store operations to the appropriate constant. We also use the element index recorded during partial interpretation to identify which struct element should be converted.

For string variables, we identify the instructions that represent string variables, create an array, and assign

3. https://github.com/SRI-CSL/whole-program-llvm

the string to the created array. Finally, we identify store instructions that use the string variable as its destination operand, and override the store instruction's source operand to use the constant string value.

Soundness of CC. LMCAS needs to avoid unsound conversions of variables to constants, which is done by detecting aliasing among stack variables. Aliasing is detected by examining the LLVM store instructions that assign values to pointer and stack variables. Our criterion for choosing to perform constant conversion for a given variable is as follows: if, after the initialization point, there is no store to that variable in the def-use chain to the neck, then the variable's value is not updated from initialization to the neck, and the variable is one on which we perform constant conversion; otherwise, the variable is left as it was in the original program.

In `wc` (Listing 1), no replacements are performed for global variables `total_lines` and `total_chars` before the neck: there are no such occurrences. Replacements are performed for referents of the pointer-to-struct `flag`: the occurrences of `flag->count_chars` and `flag->count_lines` at lines 13 and 14 are replaced with the corresponding values listed in Table 1.

3.4. Multi-Stage Simplification (MS)

This phase begins with the result of constant conversion, and performs whole-program optimization to simplify and remove unnecessary code. In this phase, we used existing LLVM passes, as well as one pass that we wrote ourselves. In particular, LMCAS uses the standard LLVM pass for constant propagation to perform constant folding; it then uses another standard LLVM pass to simplify the control flow. Finally, it applies an LLVM pass we implemented to handle the removal of unnecessary code.

Constant Propagation. This optimization step folds variables that hold known values by invoking the standard LLVM constant-propagation pass. Constant folding allows instructions to be removed.

Simplifying the CFG. LMCAS benefits from the previous step of constant propagation to make further simplifications by invoking a standard LLVM pass, called `simplifycfg`. This pass determines whether the conditions of branch instructions are always true or always false: unreachable basic blocks are removed, and basic blocks with a single predecessor are merged.

Clean Up. In the simplification pass, LMCAS removes useless code (i.e., no operation uses the result [18]) and unreachable code, including dead stack and global variables and uncalled functions. Although LLVM provides passes to perform aggressive optimization, we wrote a targeted LLVM pass that gives us more control in prioritizing the cleaning up of unneeded code, as described in Algorithm 2, which receives the modified program P_{cc} after the CC phase and the list of functions visited during the Partial Interpretation phase (`visitedFunc`).

The first priority is to remove unused functions. The goal is to remove two categories of functions: (i) those that are called only from call-sites before the neck, but not called during partial interpretation (Lines 4-6), and (ii) those that are never called from the set of functions transitively reachable from `main`, including indirect call-sites (Lines 7-10). Function removal is performed after constructing the call graph at Line 3. To handle indirect-call sites, Algorithm 2 also checks the number of uses of a function, at Lines 5 and 8, before removing the function. This check prevents the removal of a function invoked via a function pointer.

Algorithm 2: LMCAS Clean up

```
   Input: P_cc, visitedFunc
   Output: P'
1  P' ← P_cc
2  /* Remove unused functions                                   */
3  CG ← constructCallGraph(P')
4  for func ∈ CG do
5      if func ∉ visitedFunc ∧ func is not an operand of other
         instructions then
6          remove func from P' and CG
7  for func ∈ CG do
8      if func is not an operand of other instructions then
9          remove func from P' and CG
10         remove func's descendent nodes from P' and CG if they are
             not reachable from main
11 /* Remove unused Global Variables                           */
12 for var ∈ getGlobalList(P_cc) do
13     if var is not an operand of other instructions then
14         remove var from P'
15 /* Remove unused Stack Variables                            */
16 for func ∈ CG do
17     for inst ∈ func do
18         if inst is AllocInst then
19             if inst is not an operand of other instructions then
20                 remove inst from P'
21             else if inst is a destination operand of only one
                 storeInst then
22                 remove storeInst from P'
23                 remove inst from P'
```

The focus then shifts to simplifying the remaining functions. For removing global variables (Lines 12-14), we iterate over the list of global variables and remove unused variables. Finally, we remove stack variables (Lines 16-23), including initialized but unused variables by iterating over the remaining functions and erasing unused allocation instructions. (In general, standard LLVM simplifications do not remove a stack variable that is initialized but not otherwise used because the function contains a store operation that uses the variable. Our clean-up pass removes an initialized-but-unused variable by deleting the store instruction, and then the allocation instruction.)

Soundness of cleaning up: To prevent accidental removal of functions and variables, we use the LLVM `user()` API [4], which guarantees detecting whether a function/variable is used either directly or indirectly via a pointer: an item is not removed unless item.`user()` returns emptyset.

In `wc` (Listing 1), after the CC phase both the `count_chars` and `count_lines` fields of the struct pointed to by stack variable `flag` are replaced by the constants 0 and 1, respectively (see Table 1). The simplification steps remove the tests at lines 18 and 20 because the values of the conditions are always true. Because the values of the conditions in the tests at lines 17 and 19 are always false, control-flow simplification removes both the tests and the basic blocks in the true-branches. Furthermore, the removal of these basic blocks removes

4. https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains

all uses of the global variable `total_chars`, and thus the cleanup step removes it as an unused variable.

4. Implementation

Neck Miner. This component is implemented as an LLVM analysis pass. In command-line programs, we use LLVM's def/use API to track the use of argv. For configuration-file programs, we iterate over the LLVM IR code to identify call-sites for the pre-identified file-parsing APIs. The developer has the responsibility of identifying the program locations that satisfy the structural properties from §3.1.2. (i.e., locations that are executed only once, and dominate the main logic). This task is relatively easy because the developer can rely on existing LLVM analysis passes to compute the dominance tree and verify the structural properties. We argue that such efforts are manageable. More importantly, they are one-time efforts. (Such a semi-automated approach has also been used in prior work [36] and completely manual [24].) Finally, the neck location is marked by adding a special function call to the program.
Partial Interpretation. Our implementation uses KLEE [16] to perform the partial interpretation because it (1) models memory with bit-level accuracy, and (2) can handle interactions with the outside environment—e.g., with data read from the file system or over the network—by providing models designed to explore various possible interactions with the outside world. We modified KLEE 2.1 to stop and capture the set of affected variables and their corresponding values after the neck is reached. In essence, KLEE is being used as an execution platform that supports "partial concrete states." For LMCAS, none of the inputs are symbolic, but only part of the full input is supplied. In the case of word-count, the input is "wc -l", with no file-name supplied. Only a single path to the neck is followed, at which point KLEE returns its state (which is a partial concrete state). The second column in Tables 8 and 10 describes the inputs supplied to KLEE for the different examples.
Multi-Stage Simplification. We also developed two LLVM passes using LLVM 6.0 to perform constant-conversion (CC) and the clean-up step of the MS phase. We implemented these passes because of the absence of existing LLVM passes that perform such functionalities. We tried existing LLVM passes like global dead-code elimination (DCE) to remove unused code. However, global DCE is limited to handle only global variables (and even some global variables cannot be removed). We also noticed that not all stack variables are removed, so in our clean-up pass we employ def-use information to identify stack variables that are loaded but not used. Also, the removal of indirect calls is not provided by LLVM. To prevent the removal of functions invoked via a function pointer, our clean-up pass checks that the number of uses of a function is zero before removing the function.

5. Evaluation

This section presents our experimental evaluation of LMCAS. We address the following research questions:

- **Effectiveness of Neck Miner:** How *accurate* is the neck miner in identifying the neck location? (5.1)
- **Functionality Preserving and Robustness:** Does LMCAS produce *functional* programs? and how *robust* are the debloated programs produced by LMCAS? (5.2)
- **Speeding Up Debloating Process:** Does LMCAS *speed up* the debloating process w.r.t. run time? (5.3)
- **Code Reduction:** What is the debloating performance of LMCAS w.r.t. the amount that programs are *reduced in size*? (5.4)
- **Security:** Are *attack surfaces* reduced? (5.5)
- **Practicality:** Can LMCAS debloat programs based on the kinds of inputs and complex input formats (i.e., object files, binaries, and network packets) in software *used in the real world*? (5.6)

TABLE 2. BENCHMARK SETS USED IN THE EVALUATION.

Source	Label	# of apps
GNU Coreutils 8.32	Benchmark_1	15
CHISEL Benchmark	Benchmark_2	6
Tcpdump & GNU Binutils	Benchmark_3	3
wget & mini-httpd	Benchmark_4	2

Experimental Setup. Our evaluation relies on three datasets, as shown in Table 2. *Benchmark_1* contains 15 programs from GNU Coreutils v8.32 (see Table 10). *Benchmark_2* contains six programs obtained from ChiselBench (see Table 5).[5] *Benchmark_3* consists of three programs (see Table 8). The selection of programs in Benchmark_1 was motivated by their use in prior papers on software debloating. We used Benchmark_2 because it provides us a list of CVE vulnerabilities and corresponding apps; considering this dataset facilitates our evaluation of the removal of CVEs, and allows us to compare against the CVE-removal performance of Chisel and RAZOR.

All experiments were conducted on an Ubuntu 18.04 machine with a 2.3GHz Quad-Core Intel i7 processor and 16GB RAM, except the fuzzing experiment, for which we used an Ubuntu 18.04 machine with a 3.8GHz Intel(R) Core(TM) i7-9700T CPU and 32GB RAM.
Compared tools and approaches. To evaluate LMCAS, we compared it with the following tools:

- Baseline. We establish the baseline by compiling each app's LLVM bitcode at the -O2 level of optimization without applying any specialization. This baseline approach was used in prior work [53].
- OCCAM [37]. The system most comparable to the approach used by LMCAS. However, OCCAM performs only a limited amount of program optimization—i.e., standard intra- and inter-procedural constant propagation, but not loop unrolling nor more sophisticated optimizations—and thus omits a major component of partial evaluation.
- CHISEL [6] [25]. It requires the user to identify wanted and unwanted functionalities and uses reinforcement learning to perform the reduction process.
- RAZOR [46]. Similar to CHISEL, RAZOR relies on test cases to drive the debloating but incorporates heuristic analysis to improve soundness. RAZOR performs debloating for binary code, while the others operate on top of intermediate representation that is driven from source code.

5. https://github.com/aspire-project/chisel-bench
6. git #cc4fa66d2fe92e991ea19d268b65e379e675bd27

We considered CHISEL and RAZOR because they represent state-of-the-art tools that apply aggressive debloating techniques. We selected OCCAM because it is a state-of-the-art partial evaluation tool, and thus is the tool closest in spirit to LMCAS. Comparing with these various tools facilitates verifying the capabilities and effectiveness of LMCAS. Unfortunately, we could not make a head-to-head comparison with TRIMMER [53], [4] because its implementation is unavailable. Finally, for all comparisons that involve Benchmark_2, we used the same input values used in the RAZOR evaluation [46].

5.1. Effectiveness of the Neck Miner

In this experiment, we measured the effectiveness of the neck miner in facilitating neck identification. Our evaluation involved the 28 programs specified in Table 2. These programs belong to various projects: Coreutils, Binutils, Diffutils, Nginx, wget, mini-httpd and Tcpdump. For all 28 programs, neck mining was successful, and the identified neck location was used to perform debloating. Three of the programs, Nginx, wget, and mini-httpd, make use of configuration files, which are parsed using system APIs: Nginx invokes `pread` to parse the configuration file (and in exactly one place in its LLVM IR code); wget and mini-httpd use `fopen` to parse their configuration files.

For some programs, such as GNU `wc` and `date`, there were multiple candidate neck locations before the shortest-distance criterion was applied at Line 22 of Algorithm 1. (Table 12 in Appendix B gives the full set of results.)

The neck location is inside the `main` function for the majority of the programs, except `readelf` and `Nginx`. With the help of the neck miner, it took only a few minutes for the one manual step (Line 19 of Algorithm 1) needed to identify the neck locations. More specifically, for each program the analysis time for the heuristic analysis was 2 seconds on average, and it took $5-10$ minutes to perform the manual part of structural analysis. This amount of time is acceptable, given that neck identification is performed only once per program.

As mentioned in Section 3.1, the neck is identified only once for each program: the same neck can be used, regardless of what arguments are supplied. To verify this aspect, we debloated various programs based on different supplied inputs. For example, we debloated `sort` and `wc` based on 4 and 5 input settings, respectively, and (for each program) the same neck location was used with all debloating settings. Similarly, a single neck location is used for multiple debloatings for each of the programs listed in Tables 8 and 13 (Appendix B).

5.2. Functionality Preservation and Robustness

We compared the robustness of LMCAS, as well as that of CHISEL, RAZOR and OCCAM on *Benchmark_2*. We used this benchmark because it was used to evaluate both CHISEL and RAZOR, and thus the test cases used for testing those tools were available to test LMCAS; otherwise, we would have needed to come up with our own set of test cases, which would not have been trivial.

In this experiment, we ran the binaries before and after debloating against given test cases to understand their robustness. The majority of the programs debloated in *Benchmark_2* using CHISEL and RAZOR suffer from run-time issues. These issues include crashing, infinite loop, or performing unexpected operations and are reported and discussed in [46]. In our experiment, we found that all the debloated programs by CHISEL contain these issues. Among the programs in *Benchmark_2*, all but one of the OCCAM-debloated programs work correctly, and all of the LMCAS-debloated applications run correctly.

TABLE 3. EVALUATION OF FUNCTIONALITY PRESERVING AFTER DEBLOATING BY LMCAS AND OCCAM FOR PROGRAMS IN BENCHMARK_1. ✓ MEANS FUNCTIONALITY IS CORRECTLY PRESERVED; OTHERWISE: CRASHING (C), INFINITE LOOP(L), OR WRONG OPERATION (W).

Program	OCCAM	LMCAS
basename	✓	✓
basenc	✓	✓
comm	✓	✓
date	✓	✓
du	W	✓
echo	✓	✓
fmt	✓	✓
fold	✓	✓
head	L	✓
id	✓	✓
kill	W	✓
realpath	✓	✓
sort	L	✓
uniq	✓	✓
wc	C	✓

LMCAS and OCCAM have comparable results over *Benchmark_2*. To further compare with OCCAM, we debloated the programs in *Benchmark_1* according to the settings in Table 10. Five of the OCCAM-debloated programs (33%) crash (i.e., segmentation fault) or generate inaccurate results, as reported in Table 3. In contrast, all of the LMCAS-debloated programs run correctly (i.e., LMCAS preserves programs' behavior).

We gained further confidence in the robustness of LMCAS-debloated programs via fuzzing (which has also been used to assess the robustness of programs debloated by CHISEL [25]). To test whether programs debloated by LMCAS functioned correctly and did not crash, we used AFL (version 2.56b), a state-of-the-art fuzzing tool [1]. More precisely, we used afl-qemu to fuzz the debloated binary.[7] We ran afl-qemu on the debloated programs created from *Benchmark_1* and *Benchmark_2* for six days. AFL did not provoke any failures or crashes of the debloated programs in either dataset, except for `fmt` in *Benchmark_1*, which had three test cases that went over the default 1-second timeout; these timeouts are not necessarily caused by debloating. The fuzzing results including coverage are reported in Table 17 (Appendix G).

5.3. Speed of the Debloating Process

We compared the run time of LMCAS against those of CHISEL, RAZOR, and OCCAM on *Benchmark_2*. The debloating settings in this experiment are listed in Table 11 (Appendix A). As depicted in Figure 3, the run times for LMCAS and OCCAM are significantly lower than the time for aggressive debloating techniques in CHISEL and RAZOR.

7. We used the method described on the AFL GitHub repository: https://github.com/google/AFL#4-instrumenting-binary-only-apps

As a result, LMCAS runs up to 1500x, 4.6x, and 1.2x faster on average than CHISEL, RAZOR, and OCCAM, respectively. This result illustrates LMCAS substantially speeds up the debloating process in contrast to aggressive debloating tools, but also slightly outperforms partial evaluation debloating techniques.

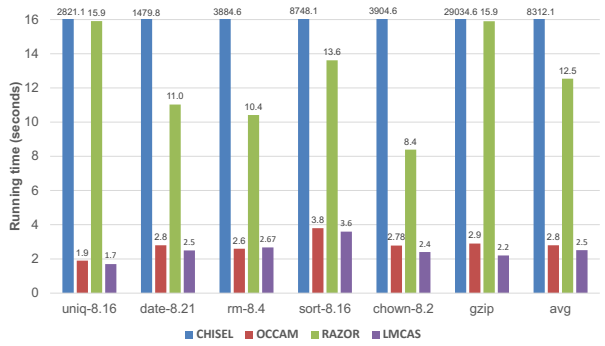

Figure 3. Running times of LMCAS, CHISEL, RAZOR, and OCCAM based on *Benchmark_2* (ChiselBench).

5.4. Code Reduction

We conducted two experiments to evaluate code reduction. The first experiment involves CHISEL, RAZOR, OCCAM, and LMCAS based on Benchmark_2, while the second uses Benchmark_1 to compare the performance of LMCAS against the baseline and OCCAM. (The characteristics of Benchmark_1 are given in Table 10 in Appendix A.) The second experiment allows us to compare reduction capabilities based on multiple size metrics, not just binary size. This comparison is not applicable for RAZOR because it operates on binary code, while CHISEL performs source-to-source debloating.

TABLE 4. REDUCTION IN BINARY SIZE

App	CHISEL	RAZOR	OCCAM	LMCAS
chown-8.2	89.15%	-42.21%	40.41%	12.18%
date-8.21	97.12%	-55.71%	45.18%	10.67%
rm-8.4	88.86%	-54.06%	41.16%	12.89%
sort-8.16	91.98%	-48.07%	43.43%	25.10%
uniq-8.16	57.62%	-58.47%	35.76%	23.90%
gzip-1.2.4	13.83%	-4.41%	2.62%	1.46%

Table 4 presents the reduction rates from the first experiment. For computing the binary-size metric, we compiled all debloated apps with `gcc` (except RAZOR, because it debloats binary code), and ran `size`. We report the sum of the sizes of all sections in the binary file (text + data + bss) because this quantity reflects the outcome of our simplifications across all sections. Interestingly, the RAZOR reduction rates are negative, which indicates that the binaries of the debloated programs are larger that the original programs. The reason is that RAZOR keeps the entire old code section, and adds the debloated code as a new code section. CHISEL achieves the best reduction results across all programs. OCCAM is always second-best, and LMCAS is always third-best. However, none of the debloated programs using LMCAS suffer from runtime issues (as discussed in Section 5.2.

Figures 4 and 5 present the results of the second experiment, where Figure 5 breaks down the "BinarySize" bars from Figure 4 on a per-application basis. The first three groups in Figure 4 show the average reductions of instructions, basic blocks, and functions in the LLVM bitcode. The bars labeled "O2" report the results from running the O2 pass, with no debloating. For the "BinarySize" bars, each original and debloated app was complied with `gcc`, and size reductions are again based on `size`. We report the sum of the sizes of all sections in the binary file (text + data + bss) because this quantity reflects the outcome of our simplifications across all sections.

LMCAS achieved significantly higher reduction rates in the binary size—around double—compared to O2. This result is due to the fact that the clean-up step of LMCAS can remove nodes in the call-graph that correspond to functions in binary libraries that are not used. Although the reduction rates of LMCAS and OCCAM are close (geometric mean binary size reduction is 25% and 22%, respectively), some of the specialized programs generated by OCCAM are not reliable (as discussed in Section 5.2).

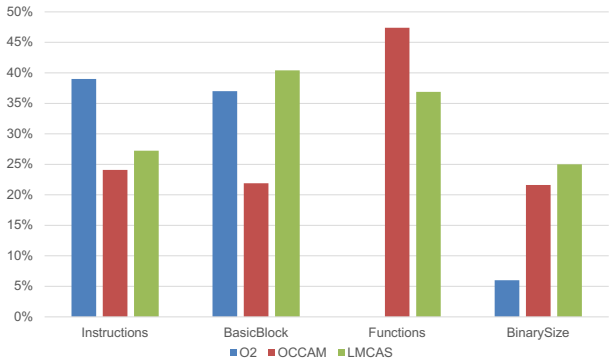

Figure 4. Average reduction in LLVM bitcode (for programs in Benchmark_1) achieved through baseline (no specialization), OCCAM, and LMCAS, using four different size metrics. (Higher numbers are better.)

Although the baseline shows a higher average reduction rate at instruction and basic block levels, its average reduction at binary size is the worse. Indeed, it increases the binary size for two programs (i.e., basenc and kill), as depicted in Figure 5 (Appendix C presents extended results), which shows the comparison results based on the reduction in the binary size that each tool achieved for each app in *Benchmark_1*.

5.5. Security Benefits of LMCAS

We performed three experiments to evaluate the capabilities of LMCAS to reduce code-reuse attacks and to remove known vulnerabilities: (i) after debloating an app, we executed the debloated app to attempt to re-run the exploit; (ii) we measured the reduction in the number of code-reuse attacks by counting the number of eliminated gadgets in the compiled version of the debloated program, compared to the compiled version of the original program; (iii) we compared the degree of gadget reduction achieved by LMCAS and LLVM-CFI (discussed in Appendix F).

Vulnerability Removal. To test the ability to mitigate vulnerabilities, we used six programs in *Benchmark_2* because this benchmark contains a set of known CVEs.

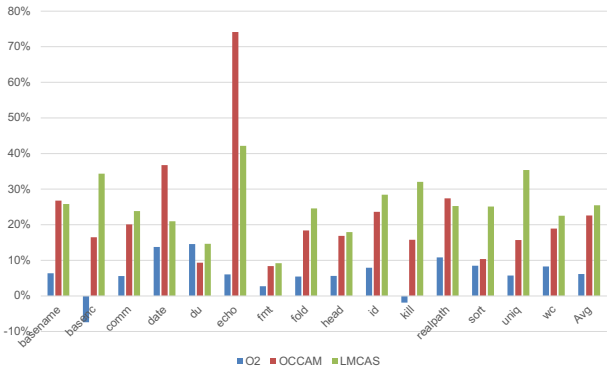

Figure 5. Binary size reduction achieved through baseline OCCAM, and LMCAS. (Higher numbers are better.)

TABLE 5. VULNERABILITIES AFTER DEBLOATING BY LMCAS AND CHISEL. ✓ MEANS THE CVE VULNERABILITY IS ELIMINATED; ✗ MEANS THAT IT WAS NOT REMOVED.

App	CVE ID	Attack Vector	RAZOR	CHISEL	OCCAM	LMCAS
chown-8.2	CVE-2017-18018	local	✓	✓	✓	✓
date-8.21	CVE-2014-9471	network	✓	✗	✗	✗
gzip-1.2.4	CVE-2015-1345	local	✓	✓	✓	✓
rm-8.4	CVE-2015-1865	local	✗	✗	✓	✓
sort-8.16	CVE-2013-0221	network	✓	✓	✓	✓
uniq-8.16	CVE-2013-0222	local	✗	✓	✓	✓

[1] CHISEL and RAZOR CVE removal is obtained from the corresponding publication.
[2] Although OCCAM removed the CVE in rm-8.4, the debloated version falls into an infinite loop at run-time.

Table 5 presents a comparison between LMCAS, RAZOR, OCCAM, and CHISEL. LMCAS and OCCAM removed CVEs from 5 of the 6 programs; however, the debloated version of `rm-8.4` produced by OCCAM falls into an infinite loop. We suspect that OCCAM may remove loop-condition checks. LMCAS could not remove the vulnerability in `date-8.21` because the bug is located in the core functionality of this program. When undesired functionality is too intertwined with the core functionality—e.g., multiple quantities are computed by the same loop—then LMCAS may not be able to remove some undesired functionality because—to the analysis phases of LMCAS—it does not appear to be useless code. In such cases, LMCAS retains the desired functionality.

Vulnerability removal for Client/Server Programs. In this experiment, we considered the two programs in *Benchmark_4*: `wget` (version 1.17.1), which is a client program, and `mini-httpd` (version 1.19), which is a server program. These programs have been used in prior work [4] to demonstrate debloating capabilities. `wget` is used for retrieving files using HTTP, HTTPS, FTP, and FTPS. `mini-httpd` is a small HTTP server that implements all the basic features of an HTTP server, including GET/HEAD/POST methods, CGI, basic authentication, and standard logging [41]. `wget` and `mini-httpd` can read configuration settings from both command-line input and configuration files. Both programs can be exploited remotely, as described in Table 6: the attack vector of these vulnerabilities is "network" according to CVSS. We debloated `wget` and `mini-httpd` with respect to the inputs shown in Table 6. We verified that functionality was not broken by running (i) the original programs with these configuration arguments, and (ii) the debloated programs, and comparing their outputs. For `wget` we compared the downloaded files; for `mini-httpd`, we ensured that the same files and headers were retrieved by both programs.

Below, we discuss the elimination of remote vulnerabilities in `wget` and `mini-httpd`.

CVE-2017-13089. The vulnerability is a heap-based buffer overflow in function `skip_short_body()` of `wget`. The severity score of the vulnerability is high (9.3 out of 10 according to CVSS[8]). By tricking an unsuspecting user into connecting to a malicious HTTP server, an attacker could exploit this flaw remotely to potentially execute arbitrary code. The function `skip_short_body()` is called when `wget` is used to send POST requests (by enabling the arguments `post-file` or `post-data`). However, if `wget` is used only to retrieve data, then `skip_short_body()` is never invoked. Debloating `wget` to disable these arguments eliminates the vulnerability.

CVE-2009-4490. This vulnerability allows data to be written to a log file without non-printable characters having been sanitized. It can allow a remote adversary to execute arbitrary commands, or overwrite files, via an HTTP request that contains an escape sequence. The severity score of this vulnerability is 5.0 out of 10 according to CVSS.[9] This vulnerability has been reported and demonstrated in several web-server programs, including `mini-httpd`, as well as `nginx` 0.7.64 and `thttpd` 2.25b0 [19]. We eliminated this vulnerability in `mini-httpd` by debloating it so that the logging feature is disabled (i.e., the flag -l is not part of the supplied inputs), which causes the corresponding logging code to be removed. In particular, debloating removed the function `re_open_logfile()`, which is responsible for opening and updating the log file.

Gadget Elimination. Code-reuse attacks leverage existing code snippets in the executable, called gadgets, as the attack payload [11]. Those gadgets are often categorized—based on their last instruction—into Return-Oriented Programming (ROP), Jump-Oriented Programming (JOP), and syscall (SYS) [47], [9]. Code-reuse attacks can be mitigated using software diversification, which can be achieved through application specialization [10]. Software debloating is a specialization technique that can both reduce the number of exploitable gadgets and change the locations of gadgets, thus diversifying the binary.

We used Gadget Set Analyzer (GSA) [11] to determine the impact of LMCAS on gadget elimination. GSA goes beyond gadget-count reduction and relies on ROPgadget [52] to collect gadget information from binaries. Thus, all derived metrics are based on the gadgets collected via ROPgadget, which has also been used in the evaluation of other debloating tools [53], [46], [25]. GSA proposes new metrics based on quality—rather than quantity—for assessing the security impact of software debloating. A detailed description of these metrics can be found in [12], [10]. All results are reported in Appendix E (Tables 14 & 15); in this section, we discuss major findings.

Table 7 summarizes the results of gadget locality, ROP-gadget introduction, and ROP-gadget removal. Gadget-locality rate is the percentage of gadgets remain-

8. https://www.cvedetails.com/cve/CVE-2017-13089/
9. https://www.cvedetails.com/cve/CVE-2009-4490/

TABLE 6. REMOTE VULNERABILITIES IN THE BENCHMARK_4 PROGRAMS, AND THE CORRESPONDING INPUTS USED FOR DEBLOATING.

App	Type	CVE ID	Attack Vector	Supplied Inputs	
				Command-line args	Configuration-file settings
wget	client	CVE-2017-13089	Network	–config=config-file	quota tries
mini-httpd	server	CVE-2009-4490	Network	-C config-file	user host port dir

TABLE 7. CODE-REUSE ANALYSIS USING GSA [10] FOR PROGRAMS IN *Benchmark_2*

App	Locality gadget rate				ROP-introduction rate				Overall ROP removal			
	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS
chown-8.2	0.0%	0.0%	0.0%	0.1%	40.2%	13.7%	78.7%	20.3%	185	96	2	38
date-8.21	0.0%	0.0%	0.0%	0.1%	21.2%	16.8%	80.4%	28.7%	198	70	-41	16
rm-8.4	0.0%	0.0%	0.0%	0.0%	37.8%	10.6%	80.6%	19.4%	206	65	-43	46
sort-8.16	0.0%	0.0%	0.0%	0.0%	42.6%	9.8%	82.3%	20.8%	264	135	-43	65
uniq-8.16	0.0%	0.0%	0.0%	0.0%	37.7%	16.5%	75.7%	19.6%	153	27	2	54
gzip-1.2.4	0.1%	6.3%	0.0%	0.0%	53.5%	28.2%	68.5%	15.9%	83	102	-73	5

ing after debloating that have not been altered and can be found at the same offset as in the original program. If debloating a program results in 0% gadget locality, then gadgets found in the original program cannot be readily used to exploit the debloated version. If debloating results in 100% gadget locality, then an exploit crafted for targeting the original program can also be used for targeting the debloated version. Our evaluation shows the gadget-locality rate is close to 0% for all compared tools across all programs. The non-0% cases are `gzip-1.2.4` using RAZOR (6.3%) and CHISEL (0.1%), and `chown-8.2` & `date-8.21` using LMCAS (both 0.1%).

The ROP-introduction rate describes the amount of new ROP gadgets in the debloated program that are not present in the original program. A low introduction rate reflects a positive impact of debloating on security. Our evaluation indicates that all debloating techniques introduce new ROP gadgets in the debloated programs. LMCAS achieves a competitive ROP-introduction rate compared to RAZOR.[10] LMCAS outperforms CHISEL across all programs (except `date-8.21`), and achieved a significantly lower introduction rate than OCCAM.

Overall ROP removal counts the number of removed ROP gadgets after debloating. The higher the number of removed ROP gadgets, the positive impact on security. Negative numbers indicate that the number of gadgets in the debloated program increased to ROP introduction. For instance, OCCAM's overall ROP removal is negative for four programs. Aggressive debloating techniques (CHISEL and RAZOR) show higher removal performance in contrast to LMCAS. However, OCCAM demonstrated the best gadget locality among all tools and for all programs. Overall, LMCAS exhibits a favorable trade-off among all metrics: reasonable ROP-gadget removal, low ROP-gadget introduction, and good locality rate.

5.6. Practicality

To evaluate the practicality of LMCAS, we used *Benchmark_3*. The programs in *Benchmark_3* have been used in prior work to evaluate scalability [37], [44], including the scalability of KLEE [15]. These programs were also used to evaluate TRIMMER [4] because they are commonly-used Linux utilities.[11]

- `tcpdump` [54] (version 4.10.0; 77.5k LOC) analyzes network packets. We link against its accompanying libpcap library (version 1.10.0; 44.6k LOC).
- `readelf` and `objdump` from GNU Binutils [7] (version 2.33; 78.3k LOC and 964.4k of library code[12]). `readelf` displays information about ELF files, while `objdump` displays information about object files.

We debloated these programs using different inputs to illustrate the capability of LMCAS to handle and debloat programs based on single and multiple inputs. For example, we debloated `readelf` based on one argument (`-S`) and nine arguments (`-h -l -S -s -r -d -V -A -I`). Table 8 breaks down the analysis time in terms of Partial Interpretation (third column) and the combination of Constant Conversion and Multistage Simplification (fourth column). For these programs, the time for symbolic execution is lower than that for the LLVM-based simplifications. This situation is expected because these programs contain a large number of functions, so a longer time is needed for the LLVM simplifications steps. The inclusion of third-party libraries diminishes the reduction rate in the binary size, which is clearly illustrated in the achieved reduction rate for `tcpdump`. Furthermore, we conducted additional experiments based on a subset of programs from *Benchmark_1* to show LMCAS's flexibility to support different combinations

TABLE 8. SCALABILITY ANALYSIS OF LARGE APPLICATIONS.

Program	Supplied Inputs	PI (sec)	CC & MS (sec)	Binary Size Reduction Rate
tcpdump	-i ens160	48.1	173.1	2%
	-i ens160 -c 5	48.2	201.7	2%
readelf	-S	10.6	41.7	4.8%
	-h -l -S -s -r -d -V -A -I	20.15	72.4	4.71%
objdump	-x	40.84	246.17	5.65%
	-h -f -p	48.07	320.11	5.71%

10. RAZOR keeps the original code section, but this section is not scanned by ROPGadget. ROPGadget scans only the added debloated code section because it becomes the loadable program segment.

11. Because deciding which lines of code belong to an individual application is difficult, lines of code (LOC) are for the whole application suites from which our benchmarks are taken. We used `scc` (https://github.com/boyter/scc) to report LOC.

12. We only consider lines in the binutils folder and the following dependencies: libbfd, libctf, libiberty, and libopcodes.

TABLE 9. BINARY-SIZE REDUCTION RATES.

Program	Version	Supplied Inputs	Binary-Size Reduction Rate	
			TRIMMER	LMCAS
readelf	2.28	-a	17.9%	5.7%
objdump	2.2.8	–D –syms -s -w	75.7%	57.1%

of supplied inputs. The selection of the supplied inputs aims to represent the core functionality of the application. Table 13 in Appendix B summarizes these results.

Although TRIMMER was unavailable to us, we compared the debloating performance of LMCAS with the results reported for TRIMMER [4, Table 1], with respect to binary size-reduction rate, for two of the programs in our *Benchmark_3* suite, `readelf` and `objdump`. For this comparison, we used the same versions, build settings, and debloating inputs that were used in the TRIMMER experiments [56]. Table 9 shows that TRIMMER has a higher reduction rate for both programs.[13] However, our comparison also revealed that LMCAS is a more general tool than TRIMMER: the TRIMMER-debloated `readelf` is specialized to handle a specific ELF file, whereas LMCAS supports delaying this input until later. TRIMMER thus has more opportunities to specialize the code with respect to that file, whereas the LMCAS-debloated program works on all ELF files supplied subsequently. The same situation holds for the TRIMMER-debloated and LMCAS-debloated versions of `objdump`.

6. Discussion and Limitations

Generality of the neck concept. This concept applies to various types of programs. Our evaluation involved 28 command-line programs; we found that the neck could be easily identified in all of the programs. We also inspected `Nginx`, `wget`, `mini-httpd` and observed that, as in command-line programs, *all* directives in the configuration file are read the main logic is reached. But this separation between configuration and main logic cannot be applied to event-driven programs, which require constant interaction with the user to handle the functionalities required by the user. Therefore, the configuration of program features is performed at various locations. However, we foresee our partitioning approach applies to event-driven programs that apply server architecture, whose life-cycle is divided into initialization and serving phases [24].

Practicality of LMCAS. The neck miner facilitates the identification of the neck. Although it involves a manual step, it provides the user with a few candidate neck locations to be inspected, from which the user can easily recognize whether a given location satisfies or violates the neck properties. For example, LLVM provides utilities for generating a CFG and a dominator tree. If a neck candidate is inside a loop in the CFG, the candidate should be discarded. Finally, neck identification is a one-time effort, thus creating a practical path to adoption. In contrast, other debloating tools, such as CHISEL [25], place a significant burden on the user because they require that the user provide a set of detailed test scripts to invoke the code that corresponds to the functionality that is intended to be preserved. This step requires understanding details of the application code, thus making it error-prone [4]. In contrast, an LMCAS user just needs to provide a command line (or configuration file) similar to the one used to run the standard version of the program, but without providing all input arguments; the user invokes the debloated program by providing the delayed inputs.

The scalability of LMCAS is influenced by KLEE (our partial interpreter), which affected the selection of programs in Section 5.6. Because KLEE has been developed to find bugs in GNU coreutils [16], its POSIX/Linux emulation layer models only a limited number of APIs (i.e., oriented towards supporting uClibc[14]). This deficiency prevented us from debloating server programs like Nginx, Apache, and Lighttpd. (For instance, ngx_cpuid() and mmap64() are required by Nginx and Lighttpd, respectively, but they are not supported by KLEE.) Rectifying this issue would have required extending KLEE to model the required operations, which is beyond the scope of this paper. In Section 5.6, we showed that LMCAS is able to handle `tcpdump` compiled with libpcap under various settings. Also, as described in Section 5.5, LMCAS could debloat the server program `mini-httpd`, using parameters taken from a configuration file.

Incorrect neck identification. Misidentifying the neck may lead to incorrect debloating. The neck miner incorporates a set of heuristics and structural features to recommend accurate neck locations. The heuristic analysis techniques aid neck identification by pinpointing the code location where the neck miner can start checking the structural properties to identify the neck. For configuration-file programs, the heuristic analysis requires the function(s) used to parse the configuration file to be identified by the LMCAS user if the parsing logic does not use system-call APIs. To make these functions visible to the neck miner, if they are located in a dynamic library, that library should be statically linked. To identify candidate neck locations, the neck miner applies a set of structural requirements that reflect the neck definition—e.g., the neck is executed only once, and the neck is an articulation point.

Precision of Constant Conversion and Cleaning up. LMCAS relies on converting a subset of the variables in the captured concrete state into constants. If one of the variables could have been converted to a constant, but was not identified by LMCAS as being convertible (and therefore no occurrences of the variable are changed), no harm is done: that situation just means that some simplification opportunities may be missed. On the other hand, if some variable occurrences are converted to constants unsoundly, the debloated program may not work correctly.

We mitigate this issue by: (1) avoiding the conversion of some variables to constants (e.g., `argv`) because it carries out the rest of inputs (i.e., delayed inputs) different than the one required by the specialized program: for instance, with `wc` the file name is not supplied during partial interpretation, while the file name is supplied to the debloated program; (2) leveraging existing LLVM APIs (i.e., `getUser`) that track the uses of variables to capture the final constant values at the end of the partial interpretation. This approach overcomes situations where a pointer indirectly updates the value of a location,

13. We learned from the TRIMMER authors that the binary-size numbers reported in [4, Table 1] were obtained via `ls -l`, which we also used for the programs debloated by LMCAS.

14. As described in https://github.com/klee/klee

and ensures that the pre-neck constant-conversion step operates using updated and accurate constant values.

Reducing the Attack Surface. LMCAS reduces the attack surface by removing some known CVEs and eliminating some code-reuse attacks. Both criteria were used in prior work on debloating [53], [25], [46], [9] to evaluate the security benefits of debloating. In this work, we conducted further analysis beyond gadget-count reduction to evaluate the elimination of code-reuse attacks. We leveraged GSA [10] to compare the quantity, quality, and locality of code-reuse gadget sets in programs debloated using LMCAS and three other tools.

7. Related Work

A variety of software-debloating techniques have been developed in the research community [29], [59], [58], [24], [46], [23], [30], [9], [6], [26], [3], [21], [32], [33]. This section discusses research on software debloating and partial evaluation related to our work on LMCAS.

Tracing Program Configurations. TRIMMER [4] tracks `argv` to annotate variables and memory objects that may hold parameters read from configuration files and/or program inputs. Rabkin et al. [49] leverages static analysis to automatically infer a program's configuration for improving a program's documentation. They focus on identifying key-value-style configurations by tracing the arguments of certain configuration APIs in the constructed call-graph. The configuration APIs are used for reading a program's configuration. Therefore, the approach finds the earliest configuration read point in a call chain. Meinicke et al. [40] propose considering disciplined or modular implementation strategies to facilitate configuration traceability. These strategies mandate separating configuration options as much as possible from other computations in the program and leveraging APIs rather than local variables. LMCAS takes advantage of the fact that such a separation is present in many programs to carry out debloating. The neck miner traces the use of `argv`, relies on heuristic analysis to identify configuration APIs, and considers the neck to be the earliest boundary candidate (i.e., at the shortest distance from the entry point).

Program Partitioning. Ghavamnia et al. [24] propose a debloating approach to reduce the attack surface in server applications. This approach partitions the execution lifecycle of server programs into initialization and execution phases. It then reduces the number of system calls in the execution phase. However, this approach requires manual intervention from the developer to identify the boundary between the two phases, but without providing certain specifications to guide the identification process. In contrast, LMCAS incorporates a neck miner to suggest a possible neck location. The neck miner provides semi-automatic support for the partitioning process and identified the neck correctly in 26 programs.

Partial Evaluation has been used in numerous domains, including debloating [53], verification [14], and generation of test cases [5]. Bubel et al. [14] use a combination of partial evaluation and symbolic execution. However, the goals and modes of interaction are different: in the work of Bubel et al., partial evaluation is used to speed up repeated execution of code by a symbolic-execution engine; in our work, symbolic execution is in service to partial evaluation by finding values that hold at the neck.

Application Specialization. For debloating Java programs, JShrink [13] applies static and dynamic analysis. Storm is a general framework for reducing probabilistic programs [22]. For debloating C/C++ programs, TRIMMER [53] and OCCAM [37] use partial evaluation. TRIMMER [53] overcomes some of the limitations of OCCAM by performing both loop unrolling and constant propagation. However, both tools perform constant propagation only for global variables, and thus TRIMMER and OCCAM miss specialization opportunities that involve local variables. That issue is addressed in a recent update of TRIMMER [4], which extends specialization to local, struct, and pointer variables (confirmed by TRIMMER's authors). It also applies taint analysis for propagating variables that potentially contain values of program-supplied configurations.

LMCAS can accurately convert the elements of struct variables into constants, and its analysis also considers pointers to base types and to struct types. A key difference between LMCAS and the most recent version of TRIMMER is the method for identifying variables influenced by supplied inputs. LMCAS applies a lightweight partial interpretation, while TRIMMER performs an expensive taint analysis, as well as an additional pass to extract information when the program uses a configuration file. The extra pass interprets various instructions that handle reading, parsing, and closing the configuration file. Moreover, TRIMMER [4] requires the user to specify the paths that lead to the call sites that parse configuration files.

Aggressive debloating tools like CHISEL [25] and RAZOR [46] can achieve a significantly higher reduction rate in the size of specialized applications; however, these tools are prone to run-time issues (e.g., crashing, infinite loops). Furthermore, the debloating process takes a long time because these tools apply burdensome techniques, based on extensive program instrumentation, and they require users to provide a comprehensive set of test cases. RAZOR uses a best-effort heuristic approach to overcome the challenge of generating test cases to cover all code. LMCAS applies lighter weight static techniques, and the specialized programs generated by LMCAS do not suffer from run-time issues.

Function Specialization. Saffire [42] specializes call-sites of sensitive methods to handle certain parameters based on the calling context. Piecewise [48] is a tool for debloating libraries; it constructs accurate control-flow-graph information at compilation and link time.

8. Conclusion

The paper presents LMCAS, a practical approach to creating specialized ("debloated") applications. LMCAS introduces the concept of the *neck*, a splitting point where "configuration logic" in a program hands off control to the "main logic" of the program. We developed a neck miner to reduce the amount of manual effort required to identify the neck. LMCAS performs partial interpretation of the configuration logic up to the neck; the main logic is then optimized according to the values obtained at the neck. LMCAS achieves substantial reductions in program size, and also reduces the attack surface.

Acknowledgment

This work is supported, in part, by a gift from Rajiv and Ritu Batra; by Facebook under a Probability and Programming Research Award; and by ONR under grants N00014-17-1-2889, N00014-19-1-2318, and N00014-21-1-2492. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2020.

[2] I. Agadakos, N. Demarinis, D. Jin, K. Williams-King, J. Alfajardo, B. Shteinfeld, D. Williams-King, V. P. Kemerlis, and G. Portokalidis. Large-scale debloating of binary shared libraries. *Digital Threats: Research and Practice*, 1(4), Dec. 2020.

[3] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 70–83, New York, NY, USA, 2019. Association for Computing Machinery.

[4] A. A. Ahmad, A. R. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, J. H. Siddiqui, and F. M. Zaffar. Trimmer: An automated system for configuration-based software debloating. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[5] E. Albert, M. Gómez-Zamalloa, and G. Puebla. Pet: a partial evaluation-based test case generation tool for java bytecode. In *PEPM '10*, 2010.

[6] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, Santa Clara, CA, Aug. 2019. USENIX Association.

[7] Binutils- gnu project - free software foundation. https://www.gnu.org/software/binutils/, 2020.

[8] P. Biswas, N. Burow, and M. Payer. Code specialization through dynamic feature observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, page 257–268, New York, NY, USA, 2021. Association for Computing Machinery.

[9] M. D. Brown and S. Pande. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.

[10] M. D. Brown and S. Pande. 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)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[11] M. D. Brown and S. Pande. 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)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[12] M. D. Brown and S. Pande. Is less really more? why reducing code reuse gadget counts via software debloating doesn't necessarily indicate improved security, 2020.

[13] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 135–146, New York, NY, USA, 2020. Association for Computing Machinery.

[14] R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Formal Methods for Components and Objects*, pages 125–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[15] F. Busse, M. Nowack, and C. Cadar. Running symbolic execution forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 63–74, New York, NY, USA, 2020. Association for Computing Machinery.

[16] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[17] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 493–501, New York, NY, USA, 1993. Association for Computing Machinery.

[18] K. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2011.

[19] hack-httpd-escape. https://www.ush.it/team/ush/hack_httpd_escape/adv.txt, Accessed Jan 2022.

[20] Nvd - vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss#, Accessed Jan 2022.

[21] N. Davidsson, A. Pawlowski, and T. Holz. Towards automated application-specific software stacks. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 88–109, Cham, 2019. Springer International Publishing.

[22] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic. Storm: Program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 729–739, New York, NY, USA, 2019. Association for Computing Machinery.

[23] M. Ghaffarinia and K. W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1009–1022, New York, NY, USA, 2019. Association for Computing Machinery.

[24] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766. USENIX Association, Aug. 2020.

[25] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 380–394, New York, NY, USA, 2018. Association for Computing Machinery.

[26] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199, 2018.

[27] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA, 1993.

[28] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[29] H. Koo, S. Ghavamnia, and M. Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, EuroSec '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] T. Kroes, A. Altinay, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida. Binrec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, FEAST '18, page 8–13, New York, NY, USA, 2018. Association for Computing Machinery.

[31] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, page 21, USA, 2005. USENIX Association.

[32] H. Kuo, J. Chen, S. Mohan, and T. Xu. Set the configuration for the heart of the os: On the practicality of operating system kernel debloating. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, pages 87–88. Association for Computing Machinery, Inc, June 2020. 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2020 ; Conference date: 08-06-2020 Through 12-06-2020.

[33] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[34] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[35] J. Li, X. Tong, F. Zhang, and J. Ma. Fine-cfi: Fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security*, 13(6):1535–1550, 2018.

[36] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1111–1128, Vancouver, BC, Aug. 2017. USENIX Association.

[37] G. Malecha, A. Gehani, and N. Shankar. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 1504–1511, New York, NY, USA, 2015. Association for Computing Machinery.

[38] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[39] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 524–533, New York, NY, USA, 2009. Association for Computing Machinery.

[40] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner. Exploring differences and commonalities between feature flags and configuration options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, page 233–242, New York, NY, USA, 2020. Association for Computing Machinery.

[41] mini-httpd. https://acme.com/software/mini_httpd/, Accessed Jan 2022.

[42] S. Mishra and M. Polychronakis. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[44] S. Poeplau and A. Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, Aug. 2020.

[45] C. Porter, G. Mururu, P. Barua, and S. Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 164–180, New York, NY, USA, 2020. Association for Computing Machinery.

[46] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, Santa Clara, CA, Aug. 2019. USENIX Association.

[47] A. Quach, A. Prakash, and L. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, Baltimore, MD, Aug. 2018. USENIX Association.

[48] A. Quach, A. Prakash, and L. Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, Baltimore, MD, Aug. 2018. USENIX Association.

[49] A. Rabkin and R. Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140, 2011.

[50] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 476–486, New York, NY, USA, 2017. Association for Computing Machinery.

[51] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, pages 409–429, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[52] Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget, 2020.

[53] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 329–339, New York, NY, USA, 2018. Association for Computing Machinery.

[54] Tcpdump/libpacap. https://www.tcpdump.org/, 2020.

[55] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, Aug. 2014. USENIX Association.

[56] Trimmer examples. https://github.com/SRI-CSL/OCCAM-Benchmarks/tree/master/examples/trimmer, Accessed Jan 2022.

[57] Gnu coreutils news. https://github.com/coreutils/coreutils/blob/ff80b6b0a0507e24f39cc1aad09d147f5187430b/NEWS#L3198, 2020.

[58] Q. Xin, M. Kim, Q. Zhang, and A. Orso. Program debloating via stochastic optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 65–68, New York, NY, USA, 2020. Association for Computing Machinery.

[59] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 160–173, New York, NY, USA, 2010. Association for Computing Machinery.

Appendix A.
Benchmark Characteristics and Debloating Settings

Table 10 lists the programs in Benchmark_1 and the supplied inputs for debloating, and mentions various size metrics about the original programs. Table 11 provides the list of supplied inputs that we used for debloating the programs in Benchmark_2. This list of inputs are obtained from [46].

TABLE 10. CHARACTERISTICS OF THE BENCHMARKS IN *Benchmark_1*.

Program	Supplied Inputs	Original			
		# IR Inst.	# Func.	# Basic Blocks	Binary Size [a]
basename	suffix=.txt	4,083	96	790	26,672
basenc	base64	8,398	156	1,461	44,583
comm	-12	5,403	110	972	32,714
date	-R	29,534	166	6,104	89,489
du	-h	5,0727	466	8,378	180,365
echo	-E	4,095	89	811	27,181
fmt	-c	5,732	115	1,095	79,676
fold	-w30	4,623	100	893	29,669
head	-n3	6,412	119	1,175	37,429
id	-G	5,939	125	1,172	36,985
kill	-9	4,539	96	898	31,649
realpath	-P	8,092	155	1,419	41,946
sort	-u	25,574	329	3,821	116,119
uniq	-d	5,634	115	1,092	37,159
wc	-l	7,076	130	1,219	41,077

[a]Total binary size obtained via the GNU `size` utility.

TABLE 11. INPUT SETTINGS FOR THE PROGRAMS IN *Benchmark_2* (OBTAINED FROM [46]).

Program	Supplied Inputs
chown	-h, -R
date	-d, –rfc-3339, -utc
gzip	-c
rm	-f, -r
sort	-r, -s, -u, -z
uniq	-c, -d, -f, -i, -s, -u, -w

Appendix B.
Neck Miner Evaluation

Table 12 describes the neck miner evaluation results. The second column indicates whether multiple neck locations are matching the control-flow properties. The third column describes if the location of the selected neck location is inside the main function.

Table 13 shows LMCAS performed debloating based on various debloating settings but using the same neck locations that has been identified in each program. This experiment shows the neck location is independent of the input arguments.

Appendix C.
Code Reduction Comparing with other tools

This section provides a detailed code reduction comparison with two more debloating approaches.

TABLE 12. NECK MINER RESULTS. THE SECOND COLUMN INDICATES IF THERE ARE MULTIPLE NECK LOCATIONS TO SELECT FROM. THE THIRD COLUMN INDICATES WHETHER THE IDENTIFIED NECK LOCATION IS INSIDE THE `MAIN` FUNCTION. THE FOURTH COLUMN INDICATES WHETHER THE PROGRAM USES A CONFIGURATION FILE.

Program	**Multiple Neck Locations**	**Inside `main`**	**Support Config file**
basename 8.32	✗	✓	✗
basenc 8.32	✗	✓	✗
comm 8.32	✗	✓	✗
date 8.32	✓	✓	✗
du 8.32	✗	✓	✗
echo 8.32	✗	✓	✗
fmt 8.32	✗	✓	✗
fold 8.32	✗	✓	✗
head 8.32	✗	✓	✗
id 8.32	✓	✓	✗
kill 8.32	✗	✓	✗
realpath 8.32	✗	✓	✗
sort 8.32	✗	✓	✗
uniq 8.32	✗	✓	✗
wc 8.32	✓	✓	✗
chown-8.2	✗	✓	✗
date-8.21	✓	✓	✗
rm-8.4	✗	✓	✗
sort-8.16	✗	✓	✗
uniq-8.16	✗	✓	✗
gzip-1.2.4	✓	✗	✗
tcpdump-4.10.0	✗	✓	✗
objdump-2.33	✗	✓	✗
readelf-2.33	✓	✗	✗
diff-2.8	✗	✓	✗
Nginx-1.19.0	✓	✗	✓
wget-1.17.1	✓	✓	✓
mini-httpd1-1.19	✓	✓	✓

TABLE 13. DEBLOATING PROGRAMS FROM BENCHMARK_1 BASED ON VARIOUS INPUT ARGUMENTS USING THE SAME NECK LOCATION IDENTIFIED IN EACH PROGRAM

App	Supplied Inputs	Required Functionality	Reduction After Debloating		
			#Func.	Binary Size	Total Gadgets
du	-b	shows number of bytes	23%	15%	46%
	-b –time	shows the time of the last modification and number of bytes	22%	14%	45%
sort	-c	check if the file given is already sorted or not	34%	28%	54%
	-n	sort a file numerically	31%	25%	51%
	-un	sort a file numerically and remove duplicate	31%	25%	51%
wc	-c	character count	42%	21%	41%
	-w	word count	42%	21%	41%
	-lc	line and character count	43%	22%	42%
	-wc	word and character count	42%	21%	42%

- Debugger-guided Manual debloating. We developed a simple but systematic protocol to perform debloating manually, which we state as Algorithm 3. The goal of this manual approach is to create an approximation for the maximum level of reduction that can be achieved by an average developer.
- Nibbler [2]. state-of-the-art tool for debloating binary code. It does not generate specialized apps, it rather focuses only on reducing the size of shared libraries. We used *Benchmark_1* to compare the performance

Algorithm 3: Debugger-guided Manual Debloating Protocol

Input: App A, Input I
Output: App A'

```
varsToRemove ← ∅
funcToRemove ← ∅
Executed ← the set of statements that GDB reports are executed, given input I
A' ← A
repeat
    for stmt ∈ A' do
        if stmt ∉ Executed then
            remove stmt from A'
            varsToRemove ← varsToRemove ∪ {stmt(vars)}
            if stmt is a call site then
                funcToRemove ← funcToRemove ∪ {func}
        while varsToRemove ≠ ∅ ∧ funcToRemove ≠ ∅ do
            for func ∈ funcToRemove do
                Remove func from funcToRemove
                if no occurrence of func exists then
                    Remove func from A'
            for var ∈ varsToRemove do
                Remove var from varToRemove
                if no occurrence of var exists then
                    Remove var from A'
        if A' does not build correctly then
            put back stmt
            undo var and func removal from A'
until no more removals of statements
```

of LMCAS against manual debloating, baseline, Nibbler, and OCCAM. Figure 6 shows the comparison results based on the reduction in the binary size that each tool achieved for each app in *Benchmark_1*. For computing the binary-size metric, we compiled all debloated apps with `gcc -O2`, and ran `size`.

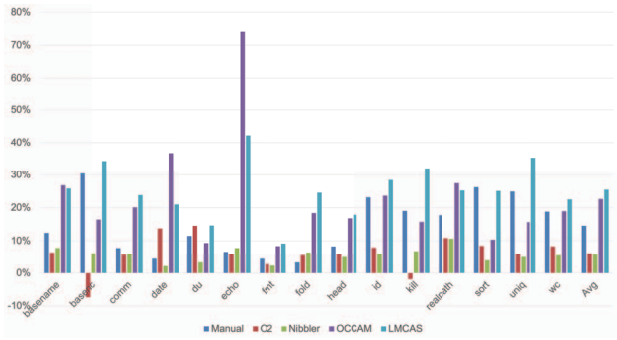

Figure 6. Binary size reduction achieved through manual debloating, baseline, Nibbler, OCCAM, and LMCAS. (Higher numbers are better.)

Appendix D. LMCAS Running Time

We measured the running time of LMCAS. Figure 7 shows the breakdown of running time, for *Benchmark_1*, between (i) partial interpretation, and (ii) Constant Conversion (CS) plus Multi-stage Simplification (MS).

The average total running time is 3.65 seconds; the maximum total running time is 13.08 seconds for analyzing `sort`; and the lowest total analysis time is 1.19 seconds for analyzing `basename`. Notably, the time for Constant Conversion and Multi-stage Simplification is low: on average, the time for constant conversion and multi-stage simplification is 0.4 seconds, while the average time for Partial Interpretation is 3.25 seconds.

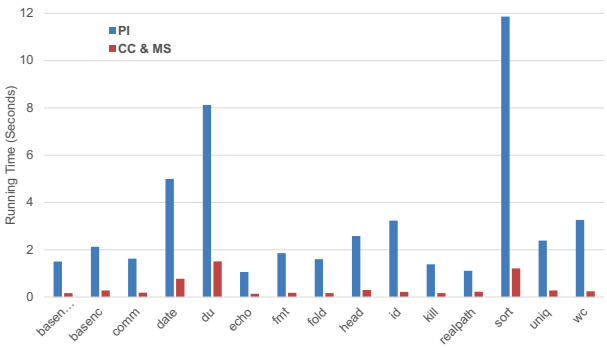

Figure 7. The time required for partial interpretation (PI) and the partial-evaluation steps (Constant Conversion and Multi-stage Simplifications) for *Benchmark_1*.

Appendix E. Gadget-Elimination Experiment

The expressivity of a set of gadgets is a measure of the computational power of a set of gadgets. GSA analyzes the gadget set with respect to three levels of expressivity: simple Turing-completeness, practical ROP exploits, and ASLR-proof. Each level requires a different number of computational classes to be satisfied. Therefore, the set of gadgets should contain at least one gadget supporting each necessary computational class to achieve a certain level of expressivity. Table 14 summarizes functional gadget-set expressivity data collected by GSA. Debloating tools were generally successful in reducing expressivity. Negative results between brackets indicate an increment in the gadget-set expressivity, which reflects introducing new gadgets that satisfied previously unsatisfied computational classes (practical ROP exploits, simple Turing-completeness, and ASLR-proof). CHISEL obtains the best results because it did not lead to any expressivity increment. RAZOR had an increase in practical ROP exploit classes for `rm` and simple Turing-complete classes for two programs (`date` & `sort`). OCCAM and LMCAS show relatively similar performance, although OCCAM increases the expressivity slightly more than LMCAS.

Table 15 summarizes an evaluation of gadgets from a qualitative perspective. The higher the score, the more difficult the gadget is to use in an exploit chain, thus a positive security impact is achieved. The numbers outside of parentheses in Table 15 represent the average gadget-quality score with respect to each attack method (ROP / JOP / COP), while numbers inside of parentheses reflect the change in gadget quality (the difference between the average quality in the original program versus the average quality in the debloated program). CHISEL succeeds in reducing the difference in average quality score of the quality gadgets for all programs except `uniq-8.16`. LMCAS and OCCAM succeed in two programs, while RAZOR achieves positive security impact in one program `date-8.21`.

TABLE 14. COMPUTATIONAL CLASSES SATISFIED (AND REDUCTION) BY GADGET SET FOR THREE FUNCTIONAL EXPRESSIVITY LEVELS

App	Practical ROP Exploit Classes				ASLR-Proof ROP Classes				Simple Turing-Complete Classes			
	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS
chown-8.2	5 (1)	7 (0)	7 (-2)	7 (0)	8 (9)	13 (9)	25 (-9)	24 (5)	6 (3)	9 (1)	8 (0)	10 (2)
date-8.21	5 (2)	7 (0)	5 (1)	7 (-1)	7 (9)	16 (1)	18 (5)	20 (-2)	5 (3)	9 (-1)	9 (-2)	7 (-2)
rm-8.4	5 (1)	8 (-1)	6 (0)	7 (0)	7 (10)	15 (5)	24 (-7)	19 (5)	3 (6)	9 (1)	7 (0)	8 (3)
sort-8.16	5 (2)	7 (0)	7 (0)	7 (1)	12 (14)	20 (2)	29 (-2)	23 (3)	6 (5)	10 (0)	9 (-1)	9 (1)
uniq-8.16	5 (2)	7 (0)	7 (-2)	7 (-1)	7 (15)	17 (0)	2 (-3)	18 (0)	5 (5)	9 (-1)	8 (-3)	4 (0)
gzip-1.2.4	5 (1)	7 (0)	7 (-1)	5 (1)	7 (1)	20 (-7)	19 (-6)	12 (3)	5 (-1)	7 (-2)	8 (-2)	6 (2)

TABLE 15. AVERAGE QUALITY SCORE (AND REDUCTION) BY ATTACK METHOD (ROP / JOP / COP).

App	Quality ROP Gadgets				Quality JOP Gadgets				Quality COP Gadgets			
	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS	CHISEL	RAZOR	OCCAM	LMCAS
chown-8.2	4.0 (-2.5)	0.93 (1.23)	1.8 (0.6)	1.4 (0.28)	1.5 (0.57)	1.0 (0.5)	1.25 (0.41)	2.06 (0.17)	1.89 (-0.04)	1.9 (-0.13)	1.39 (0.4)	1.7 (-0.05)
date-8.21	4.0 (-2.0)	1.53 (-0.1)	3.6 (-1.26)	2.0 (0.25)	1.5 (0.25)	1.5 (0.22)	1.3 (0.21)	1.4 (0.46)	1.4 (0.2)	1.7 (-0.09)	1.35 (0.26)	1.54 (-0.07)
rm-8.4	4.0 (-2.5)	0.91 (0.47)	1.9 (0.6)	1.8 (-0.03)	1.125 (0.08)	0.7 (0.17)	0.56 (1.5)	2.3 (-0.24)	1.7 (0.1)	2.0 (-0.16)	1.4 (0.4)	1.6 (-0.02)
sort-8.16	4.0 (-1.9)	1.78 (0.08)	1.7 (0.48)	1.6 (0.005)	1.5 (-0.5)	0.65 (0.8)	0.54 (1.08)	1.8 (0.38)	1.8 (0.01)	2.05 (-0.1)	1.5 (0.3)	1.9 (-0.18)
uniq-8.16	4.0 (0.75)	1.2 (0.4)	4.0 (-1.1)	2.1 (-0.2)	1.5 (-0.4)	0.0 (0.78)	0.6 (0.46)	1.3 (0.1)	1.69 (0.2)	1.8 (-0.05)	1.4 (0.4)	1.8 (-0.15)
gzip-1.2.4	5 (1)	7 (0)	7 (-1)	5 (1)	7 (1)	20 (-7)	19 (-6)	12 (3)	5 (-1)	7 (-2)	8 (-2)	6 (2)

Appendix F.
CFI Experiment

Control-Flow Integrity (CFI). CFI is a prominent mechanism for reducing a program's attack surface by preventing control-flow hijacking attacks: CFI confines the program to some specific set of CFG paths, and prevents the kinds of irregular transfers of control that take place when a program is attacked. Although CFI does not specifically aim to reduce the number of gadgets, others have observed empirically that CFI reduces the number of unique gadgets in programs [55], [35], [8]. Thus, we compared the degree of gadget reduction achieved by LMCAS and LLVM-CFI (a state-of-the-art static CFI mechanism) [8]. We compiled the LLVM bitcode of our suite of programs using `clang`, with the flags `-fsanitize=cfi -fvisibility=default`. Among the 20 programs analyzed, LMCAS outperformed LLVM-CFI on 12 programs (60%) by creating a program with a smaller total number of unique gadgets. (Table 16 in the Appendix F contains the full set of results.) The last column in Table 16 shows that a significant reduction in unique gadgets—beyond what either LMCAS or LLFM-CFI is capable of alone—is obtained by first applying LMCAS and then LLVM-CFI.

TABLE 16. COUNTS OF THE TOTAL UNIQUE GADGETS FOR THE ORIGINAL BINARIES AND THE BINARIES DEBLOATED USING LMCAS AND/OR LLVM-CFI.

Program	**Total ROP Count**			
	Original	**LLVM-CFI**	**LMCAS**	**LMCAS +LLVM-CFI**
basename	1794	964	841	261
basenc	3063	1805	1309	793
comm	2095	1145	964	794
date	12119	3654	3381	1592
du	15094	7874	8503	5873
echo	1835	446	876	442
fmt	2496	1403	1230	1158
fold	2094	1168	1015	769
head	2671	1398	1366	932
id	2514	1214	1183	801
kill	1924	1147	919	1054
realpath	3073	1610	1664	1658
sort	7558	4804	4185	3699
uniq	2516	1280	1121	776
wc	2225	1611	1320	973
objdump	115587	103241	107985	80156
readelf	58186	50512	56519	45320
tcpdump	82682	53205	67417	50809
Chown	2890	2280	2529	1998
rm	3068	2316	2579	2083

Appendix G.
AFL Fuzzing Results

Table 17 presents the results AFL fuzzing experiment including basic block coverage, crashes, and hangs. We used the tool afl-qemu-cov [15] to determine the coverage. The columns crashes and hangs report the number of test cases that caused crashing or time out, respectively.

TABLE 17. AFL COVERAGE RESULTS

program	Basic Block Coverage	unique_crashes	unique_hangs
basename	91	0	0
basenc	212	0	0
comm	118	0	0
date	260	0	0
du	949	0	0
echo	89	0	0
fmt	294	0	3
fold	152	0	0
head	137	0	0
id	173	0	0
kill	93	0	0
realpath	91	0	0
sort	619	0	0
uniq	193	0	0
wc	200	0	0

15. https://github.com/andreafioraldi/afl-qemu-cov