TRIMMER: An Automated System For Configuration-based Software Debloating

Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, Junaid Haroon Siddiqui

Abstract—Software bloat has negative implications for security, reliability, and performance. To counter bloat, we propose TRIMMER, a static analysis-based system for pruning unused functionality. TRIMMER removes code that is unused with respect to user-provided command-line arguments and application-specific configuration files. TRIMMER uses concrete memory tracking and a custom inter-procedural constant propagation analysis that facilitates dead code elimination. Our system supports both context-sensitive and context-insensitive constant propagation. We show that context-sensitive constant propagation is important for effective software pruning in most applications. We introduce *sparse constant propagation* that performs constant propagation only for configuration-hosting variables and show that it performs better (higher code size reductions) compared to constant propagation for all program variables. Overall, our results show that TRIMMER reduces binary sizes for real-world programs with reasonable analysis times. Across 20 evaluated programs, we observe a mean binary size reduction of 22.7% and a maximum reduction of 62.7%. For 5 programs, we observe performance speedups ranging from 5% to 53%. Moreover, we show that winnowing software applications can reduce the program attack surface by removing code that contains exploitable vulnerabilities. We find that debloating using TRIMMER removes CVEs in 4 applications.

Index Terms—Compilers, Partial evaluation, Program analysis

1 INTRODUCTION

To cater to a growing number of use cases, application and system developers are driven towards extending software functionality. However, usually only a small subset of all supported functionality is used in a specific deployment. The unused features can be referred to as *code bloat*. This increases the probability of residual bugs, unexpected failures, and exploitable vulnerabilities [1], [2]. Moreover, code bloat also has negative implications for resource usage. Given the trend towards edge computing (with devices having limited compute and memory), reducing bloat to improve performance and reduce resource consumption is an important goal [3], [4].

To counter the one-size-fits-all approach to software development, multiple prior studies [5], [6], [7], [8], [9], [10] have proposed techniques for *software specialization*. A target program, library, or software stack is customized with respect to constant parameter settings. This allows for removing unused code, typically through static analysis. Code debloating via software specialization can improve reliability and security since removing unused features reduces the space of possible behaviors of a program. Moreover, since software specialization employs constant folding, it simplifies program expressions and provides performance improvements [5], [6].

Software debloating is regaining research interest, with widely varied proposals using compiler-, runtime-, and machine learning-based techniques. Malecha *et al.* [8] created OCCAM, a partial evaluation tool that specializes programs with respect to pre-specified constant command-line values at compile time. OCCAM relies on standard compiler optimizations for constant propagation and code simplification. We show that standard LLVM analyses are insufficient for effective specialization. Quach *et al.* [11] developed *Piece-wise* to zero unused code in dynamically linked libraries at load time, at the cost of larger on-disk binaries (addition of a dependency graph) with no reduction in memory usage. *Chisel* [12] uses delta debugging with reinforcement learning to debloat programs, given a high-level specification of features to be retained. For each application to be debloated, a set of detailed test scripts is constructed to invoke the code that corresponds to the functionality that is intended to be preserved. This approach places a significant burden on the user, making it error-prone. In particular, it requires users to understand details of the application code [13].

1.1 TRIMMER

TRIMMER is a system that automatically specializes programs given only high-level user specifications. TRIMMER is the first code debloating system that provides (i) a simple mechanism for providing specialization specifications, in the form of application command-line inputs and configuration file contents; the specification does not require any detailed understanding of program code, thus creating a practical path to adoption, and (ii) effective specialization in reasonable analysis times, achieved by limiting expensive context-sensitive constant propagation to the program slices that are is most likely to include code conditional on configuration parameters.

The TRIMMER compilation pipeline includes both novel and traditional analyses that enable code pruning by incorporating external inputs as program constants and propagating constants through program control- and data-flow. TRIMMER includes the following compiler transforms: (i) a compiler pass to create a program variant that is specialized for constant input values (passed through command-line flags or read through configuration files), (ii) an analysis pass to identify configuration-hosting variables, (iii) constant propagation for program paths that are tainted by the detected configuration-hosting variables, and (iv) a custom loop unrolling transform that is tailored to facilitate constant propagation. Finally, TRIMMER uses standard compiler transforms for

further simplifying program expressions. TRIMMER supports both context-insensitive and context-sensitive constant propagation for configuration-hosting memory objects. For most benchmarks, we find that context-sensitive analysis provides greater code size reductions.

Recent work has argued that removing unwanted code can improve software security [11], [12]. Reducing the space of possible behaviors by a program can eliminate execution traces where vulnerabilities are exploited. Our evaluation shows that eliminating features via program specialization can remove Common Vulnerabilities and Exposures (CVEs). Proactively pruning unused code helps reduce the maintenance overhead associated with software patches and upgrades that are needed when a vulnerability is present in software features that are rarely used.

1.2 Contributions

Specifically, we make the following contributions:

- We define and use *sparse* context-sensitive constant propagation that limits analysis to the program slices that store to and read from *configuration-hosting memory objects*. Reducing the number of program variables considered for constant folding limits the amount of function cloning needed during partial evaluation.
- We show that context-sensitive analysis for all program variables usually *increases code size* due to the excessive creation of function clones during partial evaluation.
- We develop an analysis to automatically identify and annotate *configuration-hosting program variables* that *potentially* contain values read from configuration files and/or command-line inputs.
- We develop an inter-procedural file input specialization transform that lifts content from files into static values in code that are propagated by our custom constant propagation pass. Our evaluation shows that TRIMMER can effectively specialize real-world applications with respect to their configuration files.
- Our experiments demonstrate:
  Smaller binary sizes: For 20 evaluated programs, we observe a mean binary size reduction of 22.7% and a maximum reduction of 62.7%. 6 applications are specialized with respect to configuration files, and 14 are specialized for command-line arguments. In all cases, we select configurations that represent realistic usage scenarios. TRIMMER provides higher code size reduction compared to the static analysis of OCCAM, a similar tool for partial evaluation. Across our evaluated benchmarks, OCCAM provides an average code reduction of 8.6%.
  Reasonable analysis times: We show that our *sparse constant propagation* approach has reasonable overhead in terms of both analysis time and peak memory usage. Overall, we observed a mean analysis time of 12 seconds and a maximum of 20 minutes. The highest peak memory usage observed across our benchmarks was 8.7GB (*objdump*).
  Improved Performance: For 5 (out of 20) programs, we observe lower execution times ranging from 5% to 53%.
  Reduced security vulnerabilities: For 4 evaluated applications with documented CVEs (Common Vulnerabilities and Exposures), debloating commonly unused features removed a total of 5 vulnerabilities.

This journal submission is an extended version of the conference paper titled "TRIMMER: Application Specialization for Code Debloating" [14] published in the 33rd IEEE/ACM International Conference on Automated Software Engineering. The extensions to our earlier work include:

- Proposal and evaluation of *sparse* context-sensitive constant propagation. We add support for automatically annotating configuration-hosting variables,which is used to guide the *sparse* context-sensitive constant propagation.
- Support for specializing programs with respect to application-specific configuration files. The conference version of TRIMMER only supported specialization via command-line inputs.
- A section on "Attack Surface Reduction" that shows that application-specialization can remove known vulnerabilities (CVEs) in commonly used programs.
- New performance evaluation that shows that specialization can improve application performance (lower execution times).

1.3 Motivation for Code Specialization

Though modern software stacks and applications support a number of useful features, much functionality is rarely used [15]. Moreover, different users and deployment contexts are likely to exercise different non-overlapping sets of features. For instance, a DNS proxy server may allow users to make a choice between using TCP or UDP communication; it is likely that one or the other will be better suited to a particular usage scenario. In other instances, programs have a core of frequently used functionality along with auxiliary features that are less likely to be used. For instance, while the primary purpose of a webserver is to handle remote requests for content, it may include support for various application protocols and differing levels of logging, among other configurable options; such features may not be changed in practice.

Towards the goal of customizing programs with respect to their usage scenarios, one approach taken by developers (especially in the embedded systems space) is to re-implement software to create light weight counterparts of existing programs. One relevant example is the *BusyBox* software suite that includes several Unix utilities, each of which is a stripped-down version of its full-featured original counterpart. Such manual code customization requires months and years of development overhead for mid- to large-sized programs. TRIMMER aims to provide an easier and more cost-effective alternative that provides developers with an easy to use interface for specifying the deployment context via configuration files and constant command-line inputs. Code debloating via application specialization has two significant advantages:

Improving Resource Usage. TRIMMER can reduce the storage footprint and in some cases program execution times. While the code size footprint is a secondary concern in server systems with large disks, it is a significant challenge in embedded and IoT devices that contain limited flash memory (as secondary storage) [16]. Code specialization involves propagating configuration constants, which in turn facilitates compiler optimizations, such as constant folding of program expressions and strength reduction, among others. These optimizations collectively improve performance, with a more noticeable impact when the code simplification happens to be on a hot path of execution, such as a frequently run loop.

Reducing Program Attack Surface. Vulnerabilities in unused code can become active points for code reuse exploits by malicious agents. Since debloating removes unused code from the target binary, it can reduce the attack surface. Further, by specializing

away some control flow paths, the pruning can also eliminate access to exploitable vulnerabilities. Code debloating is not an alternative to existing defenses for code reuse attacks (and related exploits), but has the benefit of reducing the amount of code that needs protection by restricting the application to the specific usage scenario. From a system administrator's perspective, this translates into benefits in code maintainability. For instance, a smaller code footprint with fewer features does not require applying security patches for functionality that has been pruned from the program binary.

2 System Workflow

TRIMMER's workflow is illustrated in Figure 1. The input to the system is a *manifest file* that includes: (i) user-defined static configuration data, and (ii) the path to an LLVM intermediate representation (IR) version of the program to be debloated. The user-defined static configuration data defines the usage context of the application for a given deployment, which includes the application's command-line arguments and paths to fixed configuration files (if any). TRIMMER specializes the target application modules with respect to this usage context and generates a specialized binary executable.

TRIMMER is composed of five primary compiler transforms: (i) configuration annotations, (ii) entry-point specialization, (iii) custom loop unrolling, (iv) file input/output (I/O) specialization, and (v) custom constant propagation. The configuration annotations pass, (sparse) constant propagation for memory objects, and file I/O specialization transform are new analyses that we have developed as part of this work. Entry-point specialization is a previously employed technique for simplifying the main program function with respect to constant inputs (i.e., command-line arguments). Loop unrolling was also an existing transform; we modify it to facilitate constant folding.

The first pass in the compilation sequence is *configuration annotations*. It identifies program variables and memory objects that *may* host parameters read from configuration files and/or program inputs. It is important to identify configuration-hosting objects. These values must be aggressively propagated for effective specialization. The alternative is to propagate all possible variables. However, this can significantly increase code size as it may result in indiscriminate function cloning (function clone created for each unique calling context).

The *entry-point specialization* support incorporates user-defined static arguments into the top-level function of the target program. This, in turn, facilitates simplifying program expressions dependent on these configuration constants. The functionality is described in more detail in Section 2.2.

Next, a custom pass aggressively performs *loop unrolling* to aid later optimizations. We observe that improved loop unrolling is necessary to facilitate effective inter-procedural constant propagation. Details are discussed in Section 2.3.

To simplify applications that are configured at runtime via external files, we develop a *file specialization* transform, which evaluates file I/O library calls. The pass lifts the contents of static files (specified in a manifest) into program-level constants. Details are provided in Section 2.4.

Our custom *sparse context-sensitive constant propagation* analysis is designed to judiciously move static values through the program's call graph. More details are provided in Section 2.5.

Finally, standard compiler transforms (in LLVM) are leveraged to optimize program expressions that consume constant values.

2.1 Configuration Annotations

The *configuration annotations* analysis identifies program variables that host configuration data values. We refer to these as *configuration-hosting* objects. This LLVM pass directs our custom (inter-procedural) *sparse context-sensitive constant propagation pass*, which operates on the program slices that load or update configuration-hosting objects. The approach has two benefits: (i) it reduces analysis times by directing the analyses to execute on a subset of program paths, (ii) it avoids partially evaluating program functions that do not facilitate constant propagation. The latter eliminates unnecessary function clones that could otherwise increase code size.

We use a static analysis pass to track values that are directly or indirectly derived from the input configuration values (to be used for specialization). The goal of the analysis is to identify program variables that are tainted by the configuration data that is lifted into the program. The inter-procedural analysis propagates the taint, considering both data-flow and control-flow dependencies. The pass uses a pre-computed static value-flow graph that captures data-flow facts across a program's control flow. The graph is constructed using SVF [17], an existing framework for value-flow analysis. The resulting graph is then traversed to identify all objects that have been tainted by the configuration. Note that the source of this information includes both command-line arguments as well as data read from configuration files.

Algorithm 1: Algorithm for Configuration Annotations

```
1  Function addConfigurationAnnotations(VFG)
2      TaintSet =: [argv] ;
3      WorkList =: [argv] ;
4      repeat
5          C =: popFromList(WorkList) ;
6          ValueFlowList =: VFG.getValueFlowList(C) ;
7          repeat
8              V =: ValueFlowList.getValue() ;
9              if VFG.MemFlowSrc(V) then
10                 W =: VFG.MemFlowDst(V);
11                 TaintSet.addToList(W) ;
12                 WorkList.addToList(W) ;
13                 if W is a store instruction value then
14                     S =: getStoreDestination(W) ;
15                     TaintSet.addToList(S) ;
16                     WorkList.addToList(S) ;
17                 end
18                 if W is a call instruction to an external function F
                     then
19                     S =: getAllPointerParameters(F) ;
20                     TaintSet.addToList(S) ;
21                     WorkList.addToList(S) ;
22                 end
23                 if W is an internal function (F) call then
24                     S =: getCorrespondingFormalParameter(F) ;
25                     TaintSet.addToList(S) ;
26                     WorkList.addToList(S) ;
27                 end
28                 if W is a branch instruction with value as
                     condition then
29                     S =: getVariablesInSuccessorBlocks(W) ;
30                     TaintSet.addToList(S) ;
31                     WorkList.addToList(S) ;
32                 end
33             end
34         until not isEmpty(ValueFlowList);
35     until not isEmpty(WorkList);
36     annotateSites(TaintSet) ;
```

Algorithm 1 outlines the configuration annotations analysis. The function `addConfigurationAnnotations` takes as input the target program's value-flow graph `VFG` (that is constructed using the SVF framework). The `Worklist` (of program variables to traverse from) and `TaintSet` (of affected program variables)

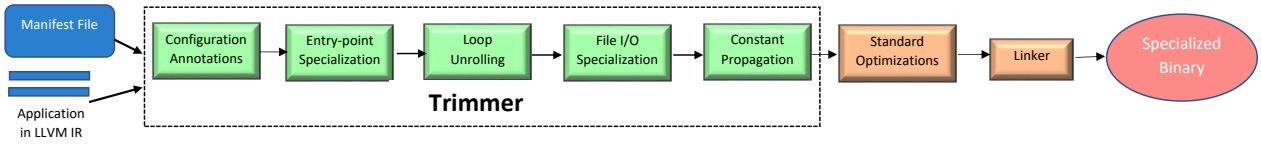


Fig. 1: TRIMMER system workflow.

is initialized with the program inputs array (which is `argv` in C/C++ programs). The outer loop (at Line 4) traverses over immediate outgoing value-flows of values in `WorkList`. The inner loop inspects each outgoing value-flow. The following actions are performed with respect to the type of value-flow:

- If the outgoing value-flow is a store instruction, the destination of the store is marked tainted.
- If the outgoing value-flow is an internal function call, the corresponding function formal parameter is marked tainted.
- If the value-flow is to an external function call, all parameters to the routine are marked tainted since it may include arbitrary store instructions that affect the corresponding memory locations (which cannot be reasoned about).
- If the outgoing value-flow is a branch instruction, the values in the successor blocks are marked tainted.

At analysis completion, the `WorkList` is empty, and `TaintSet` is populated with the variables that are tainted by the static configuration values.

Figure 2 shows the order in which the configuration annotations pass traverses the value-flow graph and marks the tainted values. The numbering adjacent to the instructions (e.g., (1), (2)) denotes the order in which the value-flow graph is traversed. Note that the examples are shown in C syntax (for readability) but actual transformations are applied on LLVM IR modules.

First, the `WorkList` is initialized with command-line input (`argv` in C/C++). Since there is a value-flow edge from `argv` to `file_path`, it is added to `WorkList` and `TaintSet`. Next, there is an inter-procedural edge from `file_path` in `main` to the `file_path` formal parameter of function `parse_config`. Further, there is a value-flow edge from `file_path` to the `fopen` function call; hence, the file pointer `fp` is tainted (and added to `WorkList`). `fp` flows to the `fgets` call, marking `str` as tainted. Eventually, the configuration-hosting variable `config` is tainted since it is accessed (stored to) in the basic blocks that are conditional on the value of `str`.

```
struct Config* config; //(4)

void parse_string(char* str){
  if(strcmp(str, "disable_logging") == 0)
     config->logging_enabled = false;
  if(strcmp(str, "disable_plugins") == 0)
     config->plugins_enabled = false;
}
void parse_config(char * file_path){
 char str[100]; //(3)
 FILE* fp = fopen(file_path , "r"); //(2)
 while( fgets (str, 100, fp) != NULL ) {
   parse_string(str);
 }
 fclose(fp);
}
void start_process(){
 if(config->plugins_enabled)
  load_plugins();
 if(config->logging_enabled)
  dump_logs();
}

int main(int argc, char *argv[]) {
 config = (struct Config*) malloc(sizeof(struct Config));
 char * file_path = argv[1]; //(1)
 parse_config(file_path);
 start_process();
 return 0;
}
```

Fig. 2: This code snippet serves as a running example. It shows values parsed from a file that are stored into a configuration-hosting object `config`.

Algorithm 2: Algorithm for Entry-point Specialization

```
1  Function entrypointSpec(manifestFile, programEntry)
2      inputArgs =: readArgs(manifestFile);
3      programArgs =: getProgramArgs(programEntry);
4      argIndex =: 0;
5      foreach argValue in inputArgs do
6          if argValue is constant then
7              replaceArgUses(argValue, programArgs[argIndex]);
8          end
9          argIndex =: argIndex + 1;
10     end
```

2.2 Entry-point Specialization

Our *entry-point specialization transform* leverages existing partial evaluation concepts [18], [19]. The transform specializes the program entry point with respect to input arguments specified in an accompanying user-provided *manifest file*. Later, our custom constant propagation pass uses this knowledge of this static data to simplify program expressions, facilitating dead code removal.

Algorithm 2 shows the algorithm for the entry-point specialization transform. The pass reads static program parameters from the manifest file. For each concretely-specified argument, all uses are replaced by the known value. Users can also mark a subset of arguments as dynamic (the values for these can be provided at runtime).

2.3 Loop Unrolling

Loop unrolling is a commonly used compiler transformation that facilitates other optimizations, such as vectorization, constant propagation, and dead code elimination. It is particularly useful for code specialization and debloating for several reasons:

- *Input parsing loops* can be unrolled to uncover program arguments that provide static input suitable for propagation and use in specialization. Usually input parsing loops are structured to handle a different argument in each iteration (e.g., `getopt` loops). Configuration variables are set based on the values obtained from arguments. To statically replace each input processing step (e.g., replacing `getopt` calls with known constant arguments), the loop must be fully unrolled to isolate

the context of each iteration (i.e., analyze each loop iteration independently). Such unrolling facilitates constant replacement, since it allows tracking of the input arguments that flow into the configuration-hosting variables.

- *Speculative loop unrolling* is used to determine whether constant propagation would be facilitated. This matters since loop unrolling without constant folding can increase code size. After a loop is completely unrolled, constant propagation is applied. If no constants are folded in the unrolled body, the loop is re-rolled into its original form. This approach stands in contrast to existing approaches (such as the standard LLVM loop unrolling support) that use static heuristics to estimate the cost and benefit of loop unrolling. Prior approaches are overly conservative and miss opportunities for profitable loop unrolling.

Algorithm 3: Algorithm for Loop Unrolling

```
1  Function fullyUnrollLoop(loop)
2      tripCount =: getLoopTripCount(loop);
3      if isConstant(tripCount) then
4          unrolledLoop =: unrollLoop(loop, tripCount);
5          runConstProp(unrolledLoop);
6      end
7      if not isConstant(tripCount) then
8          peeledLoop =: peelLoop(loop, peelCount);
9          runConstProp(peeledLoop);
10         if isNotFullyUnrolled(peeledLoop) then
11             rerollLoop(peeledLoop, tripCount);
12             unrolledLoop =: peeledLoop;
13         end
14     end
15     constsFolded =: getConstantsFolded(unrolledLoop);
16     if constsFolded == 0 then
17         rerollLoop(unrolledLoop);
18     end
```

Algorithm 3 outlines our unrolling transformation. The `fullyUnrollLoop` procedure attempts to completely unroll the provided loop. First, `getLoopTripCount` is called to get the number of iterations the loop will undergo. If this is a constant value, the loop is fully unrolled. Alternatively, if the loop trip count is non-constant, the `peelLoop` procedure is called to move the first few iterations out of the loop. (The specific number is configurable.) After unrolling, the `runConstProp` procedure runs a constant propagation pass to simplify expressions in the body of the unrolled loop.

We use a heuristic to decide whether to keep the loop unrolled or re-roll it back to the original loop. The heuristic counts the program expressions/statements that have been simplified after constant propagation. Since the purpose of the unroll transform is to facilitate propagation of *configuration-hosting* variables, we only count program expressions that involve (store or load to) the *configuration-hosting* variables. Even if a single program expression in the loop body is simplified/folded, we keep the loop unrolled.

Figure 2 shows a routine `parse_config`. It opens a configuration file (that is not shown here) and reads parameter settings using the `fgets` call. During each iteration, it reads a single line from the target file. In the loop body, `parse_string` is called to parse the file content that is read. Configuration variables are set accordingly. This loop can not be unrolled by the existing LLVM unrolling pass since the loop unroll count is non-constant. In particular, the count depends on the return value of an external function `fgets`. Since the contents of the static configuration file (specified in the manifest) are known ahead of time, the transform can compute the count of lines in the configuration file and use it as the unroll count. For this example, we assume that the configuration file includes 2 lines, with one line per configuration parameter. Hence, the `fgets` loop is unrolled with a trip count of 2. Figure 3 shows `parse_config` after unrolling has been performed.

```
void parse_config(char * filename){
  char str[100];
  FILE* fp = fopen(filename , "r");
  fgets (str, 100, fp);
  parse_string(str);
  fgets (str, 100, fp);
  parse_string(str);
  fclose(fp);

}
```

Fig. 3: Example showing a configuration file parsing loop fully unrolled using our custom loop unrolling transform.

2.4 File I/O Specialization

Configuration files include settings that define the deployment context of an application. Programs often load such information from files that are maintained by a user or an administrator. Typically, a program will parse one or more configuration files and store the extracted parameter values in (configuration-hosting) variables or other in-memory objects derived from them. This introduces an opportunity to debloat programs since significant portions of code may be invoked conditional on these values. To facilitate statically evaluating such code branches, it is necessary to lift the configuration details from the relevant file and propagate them forward. In this section, we describe our transform for specializing file operations. It replaces instructions that read from files with constant strings in the program. This introduces static input that facilitates partial evaluation.

The transform works by maintaining a *file context* for each accessed configuration file. When a file is opened, a check is performed to determine whether the path relates to a configuration. This pass assumes a list of such paths has been specified. The file context maintains a pointer to the current position as well as a Boolean attribute that indicates whether to treat the corresponding content as a constant for analysis purposes. When operations modify the context in a manner where it is no longer possible to reason about the pointer's position, the attribute is marked non-constant. Subsequent operations on the context cannot be reasoned about.

Algorithm 4 outlines the transform. The `analyzeBasicBlock` procedure is invoked on each basic block of a function. Reverse postorder is used to ensure that a block is visited after all its predecessors have been visited. The procedure is initially invoked on the first basic block of the program entry point. The argument to the procedure is a `fileContexts` data structure that maintains the state for each tracked open file. Within each basic block, file I/O calls are analyzed. Handling for the more common file I/O operations is described:

File Open: When an operation that initiates access to a file (such as `open` or `fopen`) is encountered, a context data structure is created to store the analysis state for the target file (such as the current value of the file position pointer). The analysis only maintains state for user-specified configuration files. If the mode in which the file was opened is not read-only, it is not considered for specialization.

File Seeks: When such instructions are encountered, the transform appropriately updates the file position pointer of the corresponding file context. If the offset of a `seek` can be determined via static analysis, subsequent I/O operations on the same file descriptor can be potentially specialized. If the offset is a dynamic value, the file position pointer can no longer be reasoned about. In this case, the file context is marked non-constant, which indicates that further operations on the file cannot be specialized.

File Reads: If the byte range read from the file can be statically determined, each `read` is replaced by two instructions. The first creates a constant string that corresponds to the data at the corresponding byte range in the file. The second is a `memcpy` instruction that copies this string into the target buffer. The position pointer of the file context is appropriately updated. This replacement of file read instructions introduces static data that can be further propagated with our custom constant propagation transform.

Function Calls: File descriptors may be passed as arguments to routines that access a target through I/O-related instructions, such as `mmap`, `seek`, and `read`. To support inter-procedural analysis, the current state of tracked file contexts is forwarded along with control of the analysis to the encountered callee function. The transform makes conservative assumptions regarding side-effects and marks the file context as non-constant if it is an argument to the function and the callee is externally defined (in a library).

Branches: During analysis, control is transferred to the successor basic block of a branch only after all its predecessors have been visited. This is necessary since the file context available at a particular block is computed as an intersection of the file contexts available at the predecessor blocks. For instance, if two predecessors contain unequal file seek offsets to the same file, we can no longer statically reason about the state of the file position pointer, and hence the file context is marked non-constant.

File Close: When a file descriptor is closed (by `close` or `fclose`), the context object corresponding to the file is freed.

To illustrate how the file I/O specialization pass works, we describe a small example. Figure 4 shows a file parsing routine `parse_config` that reads configuration settings. It does this in a loop where `fgets` is used to read one line of the file in each iteration. The code then populates the configuration-hosting object `config`. We assume in this example that the configuration file "config_file.txt" contains two configuration settings: "logging_disabled" and "plugins_disabled". After the transform has been applied, the result is the code shown in Figure 5. Note that the file parsing code has been simplified with respect to the configuration file settings. The parsing loop's trip count could be statically determined, allowing the loop to be unrolled. Next, each `fgets` call is replaced with the creation of a corresponding constant string and a `memcpy` that copies it to the target memory buffer `str`. The constant strings can then be used by other compiler transforms to simplify the code. While the examples are shown in C syntax for readability, the actual transformations are applied to the LLVM bitcode.

Algorithm 4: Algorithm for File Specialization

```
1  Function analyzeBasicBlock(basicBlock, fileContexts)
2      inst =: first instruction in basicBlock ;
3      repeat
4          switch inst.getType() do
5              case FileOpen do
6                  if manifestIncludesFile(filePath) then
7                      openFileContext =: createFileContext(inst);
                       fileContexts.add(openFileContext);
8                  end
9              end
10             case FileSeek do
11                 context =: fileContexts.getContext(inst);
12                 if isConst(seekOffset) then
13                     context.updateFilePosition(seekOffset);
14                 else
15                     context.markNonConstant();
16                 end
17             end
18             case FileRead do
19                 context =: fileContexts.getContext(inst);
20                 if isConst(bytesRead) and context.isConst() then
21                     readStr =: getConstString(context,
                         bytesRead);
22                     addMemcpy(readStr, targetBuffer);
23                     context.updateFilePosition(bytesRead);
24                 else
25                     context.markNonConstant();
26                 end
27             end
28             case FunctionCall do
29                 if callee is an internal function then
30                     analyzeBasicBlock(entryBB, fileContexts);
31                 end
32                 if callee is an external function then
33                     context =: fileContexts.getContext(inst);
34                     context.markNonConstant();
35                 end
36             end
37             case Branch do
38                 foreach successor block in branchInst do
39                     if all predecessors are visited then
40                         newContext =: mergePredContext();
41                         analyzeBasicBlock(successor,
                             newContext);
42                     end
43                 end
44             end
45             case FileClose do
46                 fileContext =: fileContexts.getContext(inst);
47                 fileContexts.removeContext(fileContext);
48             end
49         end
50         inst =: getSuccessorInst(inst);
51     until inst is the last instruction;
52     markVisited(basicBlock);
```

2.5 Constant Propagation

TRIMMER uses a custom inter-procedural transform that propagates the constant values in *configuration-hosting* variables and memory objects. Constant propagation of configuration values further simplifies program expressions and creates opportunities for pruning dead code.

Due to the conservative nature of constant propagation in production compilers, such as LLVM [20], only limited dead code elimination is achieved (as we demonstrate in our evaluation). Standard constant propagation passes prefer fast compilation over more precise but potentially expensive analyses. This design choice is suitable in the generic compilation pipeline but suboptimal for the specialized task of code debloating. We find that context-sensitive constant propagation is more effective when compared to the context-insensitive analysis that was supported in an earlier version of TRIMMER [14]). Both modes are supported in the current version of TRIMMER and reported on in our evaluation.

The constant propagation transform works by maintaining memory state for each of the variables marked as configuration-hosting, and tracks the loads and stores from these. Algorithm 5 describes the algorithm for the interprocedural constant propagation. The `runOnBasicBlock` procedure is invoked on each basic block of a function, in reverse postorder. Reverse postorder ensures a block is visited after all its predecessor blocks have been

```
void parse_string(char* str, struct Config *config){
  if(strcmp(str, "disable_logging") == 0)
     config->logging_enabled = false;
  if(strcmp(str, "disable_plugins") == 0)
     config->plugins_enabled = false;
}
void parse_config(struct Config *config){
 char str[100];
 FILE* fp = fopen("config_file.txt" , "r");
 while( fgets (str, 100, fp) != NULL ) {
   parse_string(str, config);
 }
 fclose(fp);
}
void start_process(struct Config *config){
 if(config->plugins_enabled)
  load_plugins();
 if(config->logging_enabled)
  dump_logs();
}
int main(int argc, char *argv[]) {
 struct Config* config = (struct Config*) malloc(sizeof(struct
      Config));
 parse_config(config);
 start_process(config);
 return 0;
}
```

Fig. 4: Configuration values are parsed from a config file and stored into a configuration-hosting object `config`.

```
const char* const_string1 = "disable_logging";
const char* const_string2 = "disable_plugins";
void parse_config(struct Config *config){
 char str[100];
 // loop is unrolled and fgets replaced with memcpy
 memcpy(str, const_string1, strlen(const_string1));
 parse_string(str, config);
 memcpy(str, const_string2, strlen(const_string2));
 parse_string(str, config);
}
```

Fig. 5: The file specialization transform replaces the `fgets` operation with `memcpy` calls to constant strings.

visited. The procedure is initially invoked on the entry basic block of the program entry routine. The argument to the procedure is a context data structure that maintains the state for each tracked memory object. Within the procedure, each instruction in the basic block is traversed. The key points of the algorithm can be summarized as follows:

Allocations: A context data structure is created for each allocation site; including both stack and heap allocations. For each allocation-site, the contiguous memory allocated is referred to as a *memory object*. For each memory object, a context data structure is created to represent the memory state of the object. The memory state for globally declared objects is created at analysis startup.

Loads and Stores: For each `Store` instruction, the memory context of the destination memory object is updated. If the source of store is a constant value, the memory state is updated with the corresponding constant value. For each `Load` instruction, the target is directly replaced with the constant value if the value to be loaded is constant. Such constant folding of loads promotes further constant propagation.

Function Calls: The `processCallInst` procedure details the policies for handling call instructions. For callee functions defined externally, the arguments and the return value of the function are marked as non-constant since the memory side-effects

Algorithm 5: Algorithm for Interprocedural Constant Propagation

```
1  Function processCallInst(callInst, context)
2      if callee is externally defined then
3          foreach argument in callInst do
4              markNonConstant(argument, context);
5          end
6      else
7          runOnBasicBlock(callee → entryBlock, context);
8      end
9
10 Function processBranchInst(branchInst)
11     foreach successor block in branchInst do
12         if all predecessors are visited then
13             newContext =: mergePredecessorContext();
14             runOnBasicBlock(successor, newContext);
15         end
16     end
17
18 Function runOnBasicBlock(basicBlock, context)
19     i =: first instruction in basicBlock;
20     repeat
21         if i is an allocation then
22             objectContext =: createObjectContext(i);
23             addToContext(objectContext, context);
24         end
25         if i is a store instruction then
26             if constant value store then
27                 updateMemContext(operand, source, context);
28             else
29                 markNonConstant(operand, context);
30             end
31         end
32         if i is a load instruction then
33             if operand is constant in context then
34                 r
35             end
36             eplaceLoadWithConstant(i, context);
37         end
38         if i is a call instruction then
39             processCallInst(i);
40         end
41         if i is a branch instruction then
42             processBranchInst(i);
43         end
44         i =: getSuccessorInst(i);
45     until i is the last instruction;
46     markVisited(basicBlock);
```

of the external function are unknown. An exception to this rule are commonly-used library calls such as file I/O calls and string processing calls (e.g., fopen, open, strcmp, strlen, atoi) which can be statically evaluated for known constant arguments. Currently, we support such static evaluation (specialization) for a total of 38 commonly-used `libc` calls.

For callee functions defined internally, the control is transferred to the callee along with the state of the memory context at the call-site. As the constant propagation transform replaces the constant memory loads inside a function body, the specialized call path is only valid in a particular memory context. In the context-sensitive mode, for each call-site with a unique set of function arguments, a new function clone is created. This is in contrast to context-insensitive mode, where at most a single function clone is created; which is possible if all call-sites have the same memory state for all the function arguments.

Branches: As branch instructions are encountered, control is transferred to the successor blocks. The `processBranchInst` shows details of handling branch instructions. Control is transferred to a successor basic block only if all its predecessors have been visited. Visiting all predecessor blocks is necessary, since the memory context available at a particular block is computed as an intersection of the constant memory contexts available at all the predecessor blocks. For instance, if one of the predecessors includes a non-constant store to a memory object, the successor's

memory context corresponding to that object is marked non-constant, regardless of a potentially constant context in a different predecessor. Similarly, if both predecessors have constant but different values for the same memory object, the successor's memory context corresponding to that object is still marked non-constant.

Figure 6 shows the code after applying our constant propagation to the code in Figure 5. `parse_config` includes 2 different calls to `parse_string` each with a different memory context. In context-sensitive mode (the default), two function clones are made, one for each context. The `strcmp` function in each specialized `parse_string` can be statically evaluated since the arguments (for which each function clone is now specialized) are constant strings. Statically evaluating the result of `strcmp` calls facilitates folding the subsequent branch instructions, thereby reducing `config->plugins_enabled` and `config->logging_enabled` to constants. By this point, the configuration-hosting variables are reduced to constants, which further allows the branches in the `start_process` function (Figure 2) to be statically evaluated; allowing for potentially pruning functions `load_plugins()` and `dump_logs()` (assuming these functions are not called from elsewhere in the program call-graph).

```
const char* const_string1 = "disable_logging";
const char* const_string2 = "disable_plugins";
struct Config* config;

// parse_string cloned and specialized for const_string1
void parse_string1(){
     config->logging_enabled = false;
}
// parse_string cloned and specialized for const_string2
void parse_string2(){
     config->plugins_enabled = false;
}
void parse_config(){
  char str[100];
  memcpy(str, const_string1, strlen(const_string1));
  // parse_string specialized for const_string1
  parse_string1();
  memcpy(str, const_string2, strlen(const_string2));
  // parse_string specialized for const_string2
  parse_string2();
}
```

Fig. 6: Context-sensitive constant propagation creates specialized clones `parse_string1` and `parse_string2` of `parse_string`. Each cloned version is specialized for a particular value of an input string being parsed.

3 Soundness of Transformations

TRIMMER provides propagation of configuration values through a combination of sequentially executed passes: configuration annotation, entry-point specialization and constant folding (includes loop unrolling, file I/O specialization and constant propagation) while allowing existing LLVM analyses to prune provably unused code in light of the introduced constants. To preserve program semantics, our transformations must prevent any incorrect constant folding (replacing a non-constant expression with a constant value). Incorrect constant folding may also potentially lead to unsound dead code elimination (eliminating functions/branches that may be invoked). To ensure the correctness of constant folding, we make conservative assumptions regarding memory side effects. While these assumptions limit the precision of our analysis, they ensure that the transformations are sound. We discuss scenarios that present threats to the validity of our transformations and the assumptions necessary to preserve soundness.

Entry-point Specialization: In the entry-point specialization transform, we replace occurrences of the program input arguments (argv references) with corresponding constant values provided in the manifest file. Entry-point specialization is a sound and incomplete transformation to specialize programs under the assumption that command line arguments are not dynamically modified in the program. Dynamically configurable software uses other means such as sockets or files for reconfiguration (e.g., HUP signal is commonly used in Unix programs to re-read the configuration file). In practice, dynamic reconfigurability is uncommon, and hence input specialization can be correctly applied in the context of most programs. Notably, we do not eliminate the command line arguments but only eliminate their references. The arguments are still present and loaded in memory. There can be pointers to argv that we cannot track due to imprecise pointer analysis but those pointers will still read the correct arguments (maintaining soundness) and these references will not be converted to constants (limiting completeness). Since only references that are guaranteed to point to arguments are replaced, other variables are not impacted. Similarly, in the constant folding transform, only the references to statically-proven constant variables are replaced, thereby not impacting variables with dynamic values.

File I/O Specialization: Similar to entry-point specialization, file I/O specialization is sound and incomplete under the assumption that the configuration file is not reloaded at runtime with new configuration parameters. (e.g., HUP signal to re-read the configuration file). To preserve correctness, we make other conservative assumptions. In scenarios where the file pointer of an open configuration file is passed as a pointer to an external function call, we assume global side-effects and no longer consider the corresponding file pointer for specialization (i.e., no further reads from the file can be folded to constants).

External Function Calls: For external function calls with unknown semantics, we make conservative assumptions regarding the memory side effects. Notably, each call argument that is not a read-only function parameter is assumed to be modified, and the corresponding context is marked as non-constant. For standard library interfaces (e.g., libc calls) with predefined standard semantics, the transformations can more precisely reason about memory side effects. To allow the tool to analyze the library calls, the users can statically link the libraries with the application modules (eliminating external calls).

Constant Folding of Global Variables. The constant folding of globals is sound under the assumption that program globals are not modified in external library modules that are not statically linked with the program. The constant propagation analysis assumes that external library interfaces do not have side effects to global variables, unless a pointer to a global is explicitly passed as argument to an external function call interface.

4 Experimental Setup

4.1 Benchmarks

We evaluate TRIMMER on a total of 20 benchmarks that include commonly-used Linux programs. Table 1 shows the list of programs used in the evaluation along with the version, lines of code (LOC), original (baseline) binary size, command-line arguments

TABLE 1: Characteristics of evaluated programs and arguments used for specialization. `config_file_path` is a placeholder for the configuration file used for specialization (with configuration options). "__" denotes dynamic-valued arguments that can be specified at runtime.

Application	Version	LOC	Command-line Arguments	Configuration File Parameters	Binary Size (-O3)	Debloated Binary Size
dnsproxy	1.17	528	-c config_file_path	authoritative authoritative-port authoritative-timeout recursive recursive-port recursive-timeout listen port user statistics	18.9KB	14.7KB
mini_httpd	1.19	3155	-C config_file_path	user host port dir	58.9KB	46.4KB
sans	0.1.0	3190	-c config_file_path	user listen test_server cn_server server	39.5KB	35.3KB
totd	1.5.3	5476	-c config_file_path -d2	forwarder port pidfile	84.7KB	76.4KB
thttpd	2.25	6733	-C config_file_path	dir nochroot pidfile port novhost host	91.9KB	83.5KB
wget	1.17.1	68K	–config=config_file_path __	quota tries passiveftp waitretry timestamping dotstyle wait dirstruct recursive	431KB	426KB
knockd	0.5	1416	-i eth0		31.9KB	27.7KB
httping	2.4	4193	-G -s -X -b -B __		58.2KB	29.3KB
bzip2	1.0.5	5295	-fkqs input_file_path		104KB	55.7KB
memcached	1.4.4	5772	-m __ -l __		84.9KB	88.9KB
aircrack-ng	1.1	5849	-b __ -a wpa -s -w dictionary.lst __		124KB	85KB
gzip	1.3.12	7013	–force –quiet __		76KB	40.8KB
netstat	1.6	7786	-a -e -p		98.4KB	57.2KB
airtun-ng	1.1	7816	-a __ -w __ __		82.2KB	52KB
netperf	2.4.3	23K	-H __ -t TCP_RR -v 0		106KB	39.5KB
readelf	2.28	72K	-a input_file_path		557.5KB	602.5KB
yices	2.61	167K	–logic=QF_AUFBV input_file_path		1.7MB	1.4MB
curl	7.47.0	174K	–compress –http1.1 –ipv4 –ssl __		170KB	170KB
gprof	2.2.8	461K	–q input_file_path		1.1MB	914KB
objdump	2.2.8	860K	–D –syms -s -w input_file_path		2.3MB	1.7MB

for which they are specialized, file arguments for applications that are specialized for static configurations files, and final debloated size.

We selected a diverse set of programs spanning different domains including webservers (*mini_httpd* and *thttpd*), compression tools (*bzip2*, *gzip*), networking (*netstat*, *netperf*, *aircrack-ng*, *airtun-ng*, *httping*), data transfer tools (*wget*, *curl*), dns servers (*dnsproxy*, *totd* and *sans*), SMT solver (*yices*), port knock server (*knockd*), memory object caching daemon (*memcached*) and commonly-used linux utilities (*objdump*, *readelf*, *gprof*). From 20 benchmarks, 6 are specialized for configuration files while 14 programs are specialized for command-line inputs.

Our selection of static program arguments is driven by two common use case scenarios. We select arguments that i) represent the core functionality of the application while leaving out the auxiliary features, and ii) represent a single use case for an application that supports multiple use cases. The first is useful for applications that provide certain core functionality, and the auxiliary features are rarely ever used, presenting an opportunity for code pruning. The second scenario is useful when applications have multiple usage scenarios. However, only a subset of those usage scenarios are likely to be used in a particular deployment.

Due to a known limitation in the LLVM-v7 loop unrolling transformation [21], we required minor changes to 3 loops in 3 different benchmarks (*mini_httpd*, *thttpd*, *yices*). The LLVM loop rotate pass (required for loop unrolling) does not properly handle loops that includes a loop condition that is composed of multiple terms. Such a loop structure at the LLVM IR level results in multiple exits to the latch (exiting) basic block. We manually rewrote such loops (only 3 across all benchmarks) to use `for` loops with the loop exit conditions added as `if` statements in the loop body with appropriately placed `break` statements. With newer LLVM versions (not yet experimented with TRIMMER) this limitation may be entirely removed.

4.2 Comparisons for Code Size Reduction.

We compare TRIMMER against:

- **Baseline -O3**: Compiling the binary without specialization and with -O3 level of optimization (optimized for performance).
- **Baseline -Os**: Compiling the binary without specialization and with -Os level of optimization (optimized for size).
- **OCCAM**: We compare TRIMMER against state-of-the-art code debloating tool OCCAM [8]. OCCAM's static analysis functionality does not support file I/O specialization, so we only provide comparisons for programs that can be specialized with command-line inputs.

  In all cases, we compare the size of the compiled binary after stripping the symbol table and debug information. TRIMMER and our baselines use LLVM version 7. OCCAM is based on LLVM version 10.

4.3 Performance and Analysis Time Evaluations.

We conducted experiments on an isolated Ubuntu virtual machine with 1.8GHz processor and 16GB RAM. For performance measurements, we use application-specific benchmarks where available and otherwise measure execution time for running the program binary with sample inputs. For *mini_httpd* and *thttpd* webservers, we use the Apache Benchmark to measure time per concurrent request. For *memcached*, we use *memcslap* to generate test workloads. For other programs, we measured execution times across 100 executions, and report the average and variance. Across all benchmarks (and across different runs), we observe a reasonably low variance of 6.1% and 6.5% in the execution times for baseline and specialized binaries, respectively.

4.4 Heuristics used to Limit Function Cloning.

Context-sensitive constant propagation is susceptible to excessive cloning which can lead to increased code size (especially with recursive functions and loops). To mitigate this, we introduce a configurable knob that sets the limit for the number of functions that can be cloned for a single function. This threshold is manually configured (and is easy to do for a developer). In our evaluations, we set this knob for 3 programs: *wget*, *gprof* and *objdump*. The optimum values we found for these 3 programs (empirically determined) are 300, 400 and 200, respectively.

4.5 Testing of Specialized Binaries

We have verified correctness of specialized binaries by performing bit-wise differential testing on the output of the original binaries and the specialized versions using a small number of carefully designed test cases. For each specialized binary produced by TRIMMER, we wrote test cases to ensure that the final debloated binaries run successfully as well as produce the same output as the original output on select scenarios.

For all of the applications evaluated, we ran the original binaries with the configuration arguments for which we specialized them and the specialized binaries, and compared their outputs. Specifically, for *bzip2*, *gzip*, *wget* and *curl*, we ensure that the correct output files (for *bzip2* and *gzip*, the compressed or decompressed files and for *wget* and *curl*, the downloaded files) are produced. For webservers (*mini_httpd* and *thttpd*), we used *wget* to ensure that they serve the same files and headers as the original unspecialized versions. For DNS servers, we generated a small number of DNS queries using *dig* and checked that identical responses are generated. For *memcached*, we ran *memcslap* to check whether the server is properly functioning or not. For *aircrack-ng*, we checked for the cracking key in a sample CAP file.

5 EVALUATION

In this evaluation, we seek to answer the following questions:

- **Q1. Impact on Code Size**: Does TRIMMER reduce code sizes for real-world applications?
- **Q2. Comparison with OCCAM**: How does TRIMMER compare to other existing tools for static analysis driven code specialization tools such as OCCAM?
- **Q3. Impact of varying Context-sensitivity**: What is the impact of varying context-sensitivity in constant propagation on code size reduction and analysis time?
- **Q4. Impact on Performance**: Can application specialization using TRIMMER improve application performance?
- **Q5. Impact on Security**: Does code debloating reduce security vulnerabilities?
- **Q6. Scalability of Analyses**: What is the peak memory usage of TRIMMER analyses?

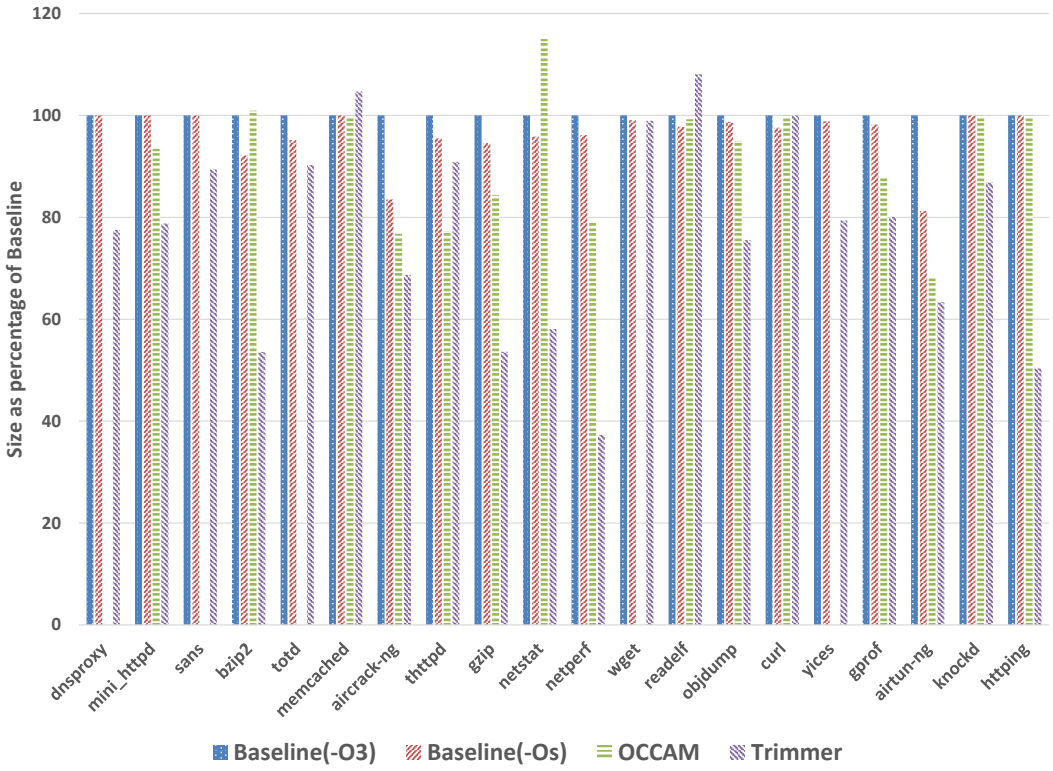


Fig. 7: Application binary sizes as a percentage of the *Baseline* after running TRIMMER. The *Baseline (-Os)* bar shows the code size reduction with -Os flag. Results with the OCCAM code specialization tool are also included.

5.1 Impact on Code Size

To answer Q1 of our evaluation questions, we evaluate the effectiveness of TRIMMER in achieving code size reductions for real-world applications. Figure 7 shows the binary size reductions after specializing programs for their corresponding chosen configurations (shown in Table 1).

We describe the programs and the corresponding chosen configurations below. We also report the percentage change in the size of binary with respect to the baselines, -O3 and -Os. For programs that had the same size for both -O3 and -Os, we provide a single value for the relative code size reduction. We also report instances where there was no code size reduction or a slight increase in code size. In scenarios where only a few program instructions are conditional on configuration constants, constant propagation of configuration values does not result in any noticeable code size reductions. Moreover, code size can also increase due to a large number of functions cloned as part of specialization.

dnsproxy is a proxy for DNS queries. The configuration file for *dnsproxy* allows for specifying parameters including ports, addresses to listen on, timeouts, and statistics (among other similar settings). We specialized it for the default configuration file that is shipped with the tool and achieve a binary size reduction of 22.4%. This configuration is an example of specialization for core functionality (scenario (i) described in Section 4.1).

mini_httpd is a small HTTP web server with basic features, including GET/HEAD/POST methods, CGI, basic authentication, and standard logging [22]. It reads user-tunable parameters from a configuration file that includes host address, port number, root directory, and logging among other options. We specialized *mini_httpd* for a static configuration that disables activity logging, virtual hosting, SSL/HTTPS and CGI, and includes username, host name, port number and root directory. While logs are useful for tracking activity, they can consume large amounts of secondary storage [23] and may not be suitable for resource constrained embedded devices with limited flash memory [24] whereas virtual hosting and CGI are auxiliary features. For this configuration, we observe a binary size reduction of 21.2%.

sans is an anti-spoofing DNS server [25] that supports both TCP and UDP connections. *sans* can be configured to use a SOCKS5 connection or a plain TCP connection. *sans* reads from a configuration file that includes parameters for configuring host address and DNS server. We specialized *sans* for SOCKS5 support in one configuration and for plain TCP support in the other. This configuration is an example of a scenario where only one of multiple configurations is likely to be used in deployment, hence presenting an opportunity to specialize (scenario ii) described in Section 4.1). Across both specialized binaries, different functionality is pruned but (incidentally) we get the same binary size reduction of 10.6%.

totd is a DNS proxy nameserver that facilitates IPv6-only networks and hosts to communicate with IPv4 connections using network or transport level translation mechanisms [26]. *totd* reads configuration options via a configuration file with options including service port, list of interfaces (to listen on), and enabling/disabling support for IPv4 and/or IPv6 queries. We specialized the program for two configurations, one in which *totd* listens to client queries on IPv6 addresses and the other in which it only listens on IPv4 addresses. For both configurations, we achieved a binary size reduction of 9.8% and 5.2% compared to -O3 and -Os baselines, respectively. This configuration is also an example of specializing for common use case scenarios.

thttpd is a simple HTTP server [27] that is suitable for embedded devices. We specialized *thttpd* for parameters including the host name, the used port number, root directory (among similar options), while disabling support for logging, chroot and virtual hosting. This specialization is for the core use case of *thttpd*, and results in a binary size reduction of 9.1% and 4.8% compared to -O3 and -Os baselines, respectively.

wget is a package for retrieving files with a variety of supported protocols including HTTP, HTTPS, FTP, FTPS. *wget* can read configuration settings from both command-line input and configuration files. The configuration options include configuring progress bars, rate limiting, enabling/disabling debug output and selecting IP address family among many other options. We specialized *wget* with a configuration that includes quota, number of retries, passive FTP mode, time-stamping, wait between retries, retrieval output, wait between connections, creating directory structure and recursive mode excluding secondary options such as header, proxy, recursive level and encoding: achieving only minor binary size reductions of 1.1% and 0.1% compared to -O3 and -Os, respectively.

knockd is a port-knock server. It listens to all traffic on an Ethernet (or PPP) interface, looking for special "knock" sequences of port-hits. We specialized *knockd* for a common use case using its default configuration for listening on an Ethernet interface (disabling support for the PPP interface). Specializing for this configuration achieves a binary size reduction of 13.2%.

httping tests and measures the latency and throughput of a webserver. We specialized *httping* to use GET requests for downloading compressed data from a target server and excluded auxiliary features such as options for configuring timeout thresholds, SSL based connections, support for reading from a configuration file, and support for a SOCK5 proxy server. This configuration results in a binary size reduction of 49.6%.

bzip2 is an open-source file compression program [28]. We created two specialized versions of *bzip2*, one for compression and the other for decompression. Creating separate specialized versions is useful in scenarios where there is unidirectional communication. For example, it is not uncommon that data accumulated on IoT devices is compressed before sending to the cloud (to reduce network bandwidth usage), and decompressed on the receiving cloud server [29]. In such a scenario, the IoT device only needs compression functionality, while the cloud server only needs decompression functionality. Specialization of *bzip2* for compression resulted in binary size reduction of 46.4% and 41.8% compared to -O3 and -Os baselines, respectively. Specialization for decompression resulted in reductions of 54.3% and 50.4% compared to -O3 and -Os baselines, respectively.

gzip is a relatively older compression program that uses Lempel-Ziv (LZ77) coding [30]. Specializing *gzip* for compression provides a binary size reduction of 46.4% and 43.3% compared to -O3 and -Os baselines, respectively. Specializing *gzip* for decompression provides a binary size reduction of 8.3% and 3.1% compared to -O3 and -Os baselines, respectively.

memcached is a popular key-value distributed memory object caching system. It is used extensively for caching results of database queries and API calls [31]. We specialized *memcached* for two essential arguments: maximum memory for object storage (-m) and the IP address to bind to (-l). All other configurations (e.g., data item sizes, page sizes) are set to their default values. For this configuration, we observe a slight code size increase of 4.8%.

aircrack-ng is a WiFi key cracking program that supports two encryption technologies, WEP and WPA/WPA2-PSK. We specialized *aircrack-ng* for cracking WPA/WPA2-PSK keys (removing support for WEP keys) and added an option to display the cracked key (if found), which is a common use case scenario. This configuration results in binary size reductions of 31.3% and 17.7% compared to -O3 and -Os baselines, respectively.

netstat prints information about the Linux networking subsystem. We specialize *netstat* with -all option, showing all active internet connections and UNIX domain sockets. With this option, we achieve binary size reductions of 42% and 39.4% compared to -O3 and -Os baselines, respectively.

airtun-ng is a virtual tunnel interface creator. We specialize *airtun-ng* with a specified MAC address for the access point and a WEP key to encrypt packets, excluding support for other auxiliary features (e.g., option to replay packets). With this configuration, we achieve binary size reductions of 36.6% and 33.3% compared to -O3 and -Os, respectively.

netperf is a network performance benchmark that can measure various aspects of networking performance. We specialize *netperf* for a commonly-used network test TCP_RR (TCP request/response), and remove support for other tests such as TCP_STREAM, UDP_STREAM and UDP_RR. This configuration represents the scenario where a program/tool supports multiple use cases but the deployment is likely to use only one specific setting. This configuration results in binary size reductions of 62.7% and 61.2% compared to -O3 and -Os baselines, respectively.

readelf displays information about one or more ELF format object files. The options control what particular information to display. We specialize *readelf* for –all option, which displays information contained in ELF header, file segments headers, file section headers, symbol table, file relocation section, dynamic section, notes segments and/or sections and version section. We leave extra details such as section information, unwind section information and architecture-specific information. Like *memcached*, specializing *readelf* for this configuration results in a slight code size increase of 8%.

yices [32] is an SMT solver that includes support for multiple logics (e.g. propositional, bitvector etc.) We specialized *yices* for arrays, bit vectors and uninterpreted functions, while disabling support for the logic of linear and non-linear arithmetic. This configuration is useful for tasks such as hardware verification [33] and symbolic execution [34]. For this configuration, we achieve binary size reductions of 20.6% and 19.7% compared to -O3 and -Os, respectively.

curl [35] is a tool commonly used for transferring data. While *curl* is used in a variety of different ways, we specialized it for the common usage scenario of reading data over an HTTPS connection given a target URL. This particular *curl* configuration is commonly used for reading RSS feeds. For this configuration, the code size remains the same (no increase or decrease).

gprof produces an execution profile of C, Pascal, or Fortran77 programs. *gprof* takes as input profile data dumped by a profile-instrumented program that is compiled with the -pg option (supported in *clang* and *gcc*). It provides the following kinds of profile analysis outputs: (i) a flat profile that shows the amount of time spent in each function, (ii) a call graph mode that shows the program call graph with annotations on the time spent in each function and its children, (iii) an annotated source listing mode that labels each program instruction with the number of times it

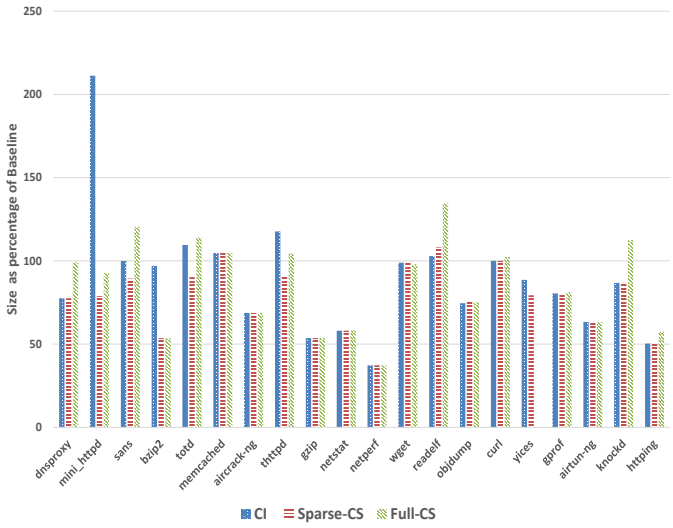


Fig. 8: Across benchmarks, comparing binary size reductions for context-insensitive analysis (*CI*), context-sensitive analysis just for *configuration-hosting* variables (*Sparse-CS*), and context-sensitive analysis for all program variables (*Full-CS*).

is dynamically executed. We specialize *gprof* for the call graph profile mode (option (ii)), and achieve binary size reductions of 19.9% and 18.4% compared to -O3 and -Os, respectively.

objdump is a commonly-used Linux utility for extracting information of binary object files. We specialized *objdump* for displaying disassembly of all program sections (code and data sections) of an object file. This configuration excludes features for printing out debugging information and printing dynamic relocation entries (among other such auxiliary features). For this configuration, we achieve binary size reductions of 24.5% and 23.5% compared to -O3 and -Os, respectively.

Summary. For 14 benchmarks, we get code size reductions of more than 10%, minor reductions for *wget*, *thttpd*, and *totd*, no code size change for *curl* and a slightly increased binary size for *readelf* and *memcached*. Across benchmarks, TRIMMER provides a mean code size reduction of 22.7% with a maximum reduction of 62.7% compared to the -O3 baseline. Compared to the -Os baseline TRIMMER provides a mean code size reduction of 20.4% and a maximum reduction of 61.2%. These results demonstrate that TRIMMER can reduce code sizes for real-world applications.

5.2 Comparison with OCCAM

We compare TRIMMER with OCCAM's default static-analysis-based functionality [8] to answer Q2 of our evaluation questions. Since this usage of OCCAM does not support file I/O specialization, we could not evaluate applications that strictly required a configuration file for purposes of specialization (wget, dnsproxy, totd, and sans). Moreover, we do not compare against *yices*, since its specialized binary (after running OCCAM) exhibited runtime errors. We compare the remaining 15 out of 20 programs in our benchmark set. Similar to the knob we support for limiting maximum applicable function cloning (in context-sensitive constant propagation), OCCAM supports four modes for intra-module specialization. These include `only-once` (i.e. a single function can be cloned at most once), `aggressive` (i.e. a new function clone created per unique calling context), `nonrec-aggressive` (i.e. same as aggressive mode but does not clone recursive functions), and `bounded` (i.e. a single function can be cloned at most k times, where k is a configurable parameter). We evaluated resultant binaries with all four modes of intra-module specialization in our experiments. For bounded mode, we tried different values of $k = 1, 3, 5, 10, 50, 100, 200$. For all applications, the `only-once` policy resulted in the smallest code sizes. Figure 7 shows the reduction with OCCAM in the `only-once` mode. Compared to OCCAM (in the `only-once` mode), we found that TRIMMER provides higher code size reductions for 12 of the 15 evaluated benchmarks. On average, OCCAM only provides 8.6% code reduction whereas TRIMMER provides a reduction of 26% across these 15 programs.

Also, OCCAM supports coarse-grained control on function cloning. For instance, the `only-once` policy is overly conservative because often multiple function clones are required when different calling contexts introduce new unique constant values. The `aggressive` mode is overly aggressive in that it creates unnecessary function clones. The sparse context-sensitive specialization mode in TRIMMER enables a sweet-spot where multiple function clones are enabled for configuration-hosting variables to facilitate constant propagation, and avoided/skipped for functions that do not involve configuration values.

5.3 Impact of Varying Context-Sensitivity

We next answer Q3 of our evaluation questions by evaluating the impact of varying context-sensitivity in constant propagation on code size reduction and analysis time. We evaluate three different analysis modes:

- *CI:* context-insensitive analysis for *all* program variables.
- *Sparse-CS:* context-sensitive analysis for *configuration-hosting* variables. This is the default in TRIMMER.
- *Full-CS:* context-sensitive analysis for *all* program variables.

Impact on Code Size Reductions.

Figure 8 illustrates the code size reductions achieved across benchmarks while varying context-sensitivity; specifically three modes: CI, Sparse-CS, and Full-CS. *yices* is not shown for Full-CS since the analysis runs out of memory after 5 minutes. For 6 programs (*memcached*, *netstat*, *netperf*, *aircrack-ng*, *airtun-ng* and *gzip*), CI, Sparse-CS, and Full-CS exhibit similar reductions. For 9 programs, Sparse-CS provides the highest reduction in code size. For the remaining 5 programs (*wget*, *bzip2*, *objdump*, *httping*, and *readelf*), Sparse-CS is a close-second in terms of reductions. For 3 programs (*mini_httpd*, *bzip2*, and *thttpd*), CI performs worse compared to both Sparse-CS and Full-CS. This is because the imprecision of context-insensitive analysis leads to missed opportunities for constant folding (and often loop unrolling based on constant trip counts) eventually impacting the amount of achievable debloating. Overall, Sparse-CS appears to find a sweet-spot between CI and Full-CS since a) it allows for effective constant propagation via context-sensitive analysis, but also b) is conservative in function cloning since it only considers configuration-hosting variables. Overall, Sparse-CS provides an average reduction of 22.7%, Full-CS provides an average reduction of 14.1%, and CI provides an average reduction of 10.9%. Hence, Sparse-CS outperforms in reducing code size.

Analyzing Function Cloning and Dead Function Removal. Table 2 displays the functions cloned (FC), instructions cloned (IC), and functions removed (FR) and instructions removed (IR) for each benchmark. On average, CI clones 51.3 functions while removing 187.1 functions. Sparse-CS clones 101.1 functions while

TABLE 2: Statistics on Functions Cloned (FC), Instructions Cloned(IC), Functions Removed (FR) and Instructions Removed (IR) for all three modes CI, Sparse-CS, and Full-CS.

	CI				Sparse-CS				Full-CS			
Program	**FC**	**IC**	**FR**	**IR**	**FC**	**IC**	**FR**	**IR**	**FC**	**IC**	**FR**	**IR**
dnsproxy	3	87	2	349	1	75	2	349	20	230	2	206
mini_httpd	4	10478	1	-8424	23	378	17	2793	24	1539	17	1673
sans	3	435	6	1708	11	239	7	1931	93	1094	8	1518
bzip2	17	1247	14	3486	21	179	42	12225	66	929	47	12248
totd	29	1600	16	3984	11	292	16	4964	123	2089	16	3584
memcached	7	2929	11	1424	6	2927	11	1424	7	2929	11	1424
aircrack-ng	3	128	39	7141	25	846	39	6423	31	897	39	6372
thttpd	32	2690	5	9	70	533	10	4226	150	3159	10	1701
gzip	35	530	56	7438	13	431	56	7379	1126	91	56	7381
netstat	1	8	152	6708	0	0	152	6708	1	8	152	6708
netperf	20	1080	51	10849	16	1236	51	10799	39	1698	51	10523
wget	41	1298	81	16294	184	3730	89	14400	191	3813	89	14323
readelf	57	8225	54	18790	154	8921	54	18235	728	22150	54	5006
objdump	24	197	1009	150929	103	2747	1009	148355	119	1394	1009	149732
curl	23	780	16	5222	939	805	16	5034	5293	1473	16	4499
yices	670	11925	1761	43860	31	228	2470	65318	-	-	-	-
gprof	45	3339	366	91084	48	1794	366	92017	493	4573	366	89869
airtun-ng	4	91	61	6438	3	85	61	6444	4	91	61	6438
knockd	3	672	12	1108	7	773	12	1007	52	1302	12	478
httping	4	140	28	4209	356	165	28	4143	2992	373	28	3988

removing 225.4 functions. Full-CS clones 608 functions while removing 107.5 functions. It can be noticed that Sparse-CS removes more functions than CI and Full-CS and clones 2x more than CI but 6X less than Full-CS. Another useful metric here is the ratio of instructions cloned to instructions removed (IC/IR) which can be used to measure how profitable instruction cloning is across the three analysis modes (CI, Sparse-CS, Full-CS). On average across benchmarks, CI requires 0.13 instruction clones to remove 1 instruction, Sparse-CS requires 0.06 and Full-CS requires 0.15. The (IC/IR) stats further confirms our hypothesis that context-sensitive analysis for a subset of variables reduces the amount of instruction clones required while achieving similar code removal. The high (IC/IR) ratio for Full-CS shows that unnecessary high count of instruction clones were created during specialization.

Impact on Analysis Time. The overall tool analysis time is mostly spent in 2 phases: *configuration annotations* and *constant propagation* (other phases have low cost). Figure 9a shows analysis times for *configuration annotations* pass on each benchmark. Note that this pass is independent of the level of context-sensitivity, since configuration annotations would be the same for each mode.

For 13 programs (relatively smaller-sized in terms of LOC), this step is fast; consuming less than 30 seconds. The largest three programs (*gprof*, *yices*, and *objdump*) show a non-linear increase in analysis-time with increasing program size. The reason for the non-linear increase is the cubic time complexity of Anderson's pointer analysis [36] which is used in the value-flow graph construction phase. Overall, the configuration annotation analysis consumes a (geometric) mean execution time of 29 seconds, with a minimum of 0.7 seconds (*dnsproxy*) and a maximum of 2482 seconds (*objdump*).

Figure 9b compares the analysis time for our custom constant propagation pass across the three modes CI, Sparse-CS, and Full-CS. For 6 programs, CI provides the lowest analysis times, while for the other 14 programs, Sparse-CS is the fastest. Full-CS is slowest in all cases (except *mini_httpd*, *thttpd* and *curl*; where CI is the slowest). Across benchmarks, CI has a (geometric) mean analysis time of 19.2 seconds, with a minimum of 0.50 seconds (*knockd*), and a maximum of 3800 seconds (*curl*). Sparse-CS has a geometric mean analysis time of 12 seconds, with a minimum of 0.40 seconds (*knockd*), and a maximum of 1210 seconds (*curl*). Full-CS analysis has a geometric mean of 31 seconds, with a minimum of 0.60 (*knockd*), and a maximum of 2693 seconds (*curl*). Hence, Sparse-CS analysis performs on average 1.6x faster than CI, and 2.6x faster than Full-CS. These results show that Sparse-CS has comparable analysis times to CI, while providing significantly improved code size reductions (22.7% compared to 10.9%).

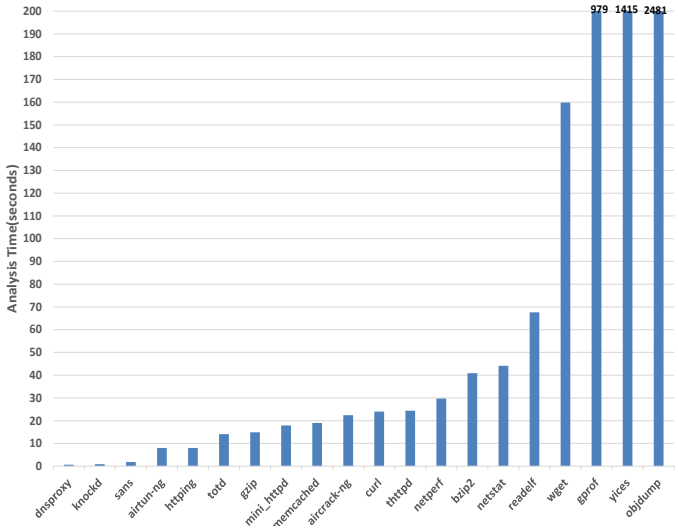
Overall, we find that program size is only loosely related to analysis times. The analysis time is dependent on a number of factors including (but not limited to): a) program paths visited in the analysis, b) number of control-flow merges that require merging the memory context, and c) amount of loop unrolling (which can increase cost for other transforms).

5.4 Impact on Performance.

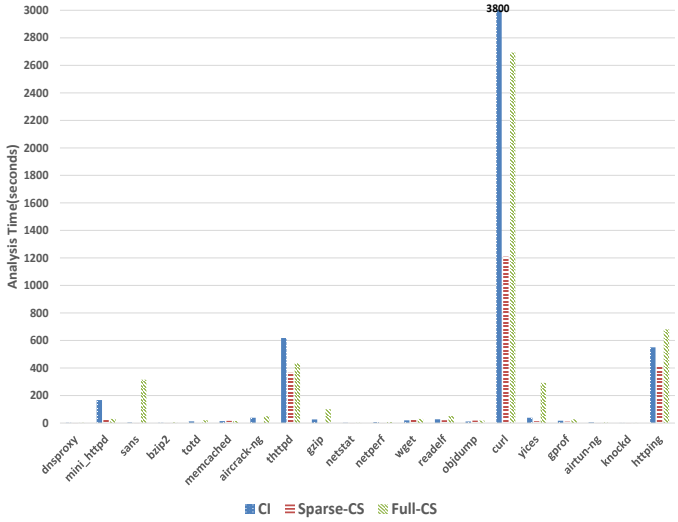
To answer Q4 of our evaluation questions, we measure the effect of program specialization on its performance. Figure 10 shows the program performance after specialization compared to the performance of the unspecialized program (unspecialized baseline is set at 100% in the Figure). For 5 (out of 20) benchmarks, we observe performance improvements greater than 5%, for 2 (*aircrack-ng*, *netperf*) benchmarks, TRIMMER achieves improvements higher than 20%, with the highest speedup of 53% (*aircrack-ng*). These speedups are a result of optimizing the hot path of execution due to code optimizations enabled by specialization. For instance, constant folding in hot program loops can reduce loop processing times; resulting in faster execution. For the rest of the programs, we either see no noticeable changes, or very minor slowdowns (possibly due to the unpredictable nature of how compiler transforms interact).

5.5 Impact on Security

We believe that debloating is consistent with improving software security. Reducing functionality limits the potential attack targets in the application [11], [12]. If a vulnerability is discovered in a piece of code, system administrators need to apply one or more



(a) Time taken for configuration annotations pass.



(b) Time taken for constant propagation pass in *CI*, *Full-CS*, and *Sparse-CS* modes

Fig. 9: Analysis time for the configuration annotations pass and constant propagation pass. Most of the overall analysis time in TRIMMER is consumed in these two passes.

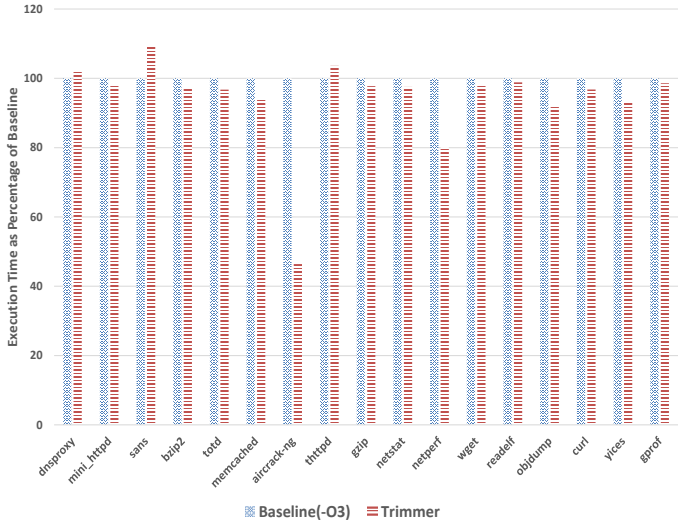


Fig. 10: Benchmark performance improvements after TRIMMER specialization.

software patches. When these updates are applied to networked systems, the resulting downtime can impose significant costs at both the client users and the service vendors [37]. Moreover, when the software in question is deployed on devices in the field without network connectivity, updating it may require a human to visit the installation, a costly proposition. In contrast, if the program had been debloated, the vulnerability may have been in functionality that was eliminated. In such cases, the cost of updating does not need to be incurred. While debloating does not eliminate the need for existing security defenses, it reduces the amount of code that needs protection. Moreover, restricting an application to specific usage scenarios reduces the possible behaviors of a program. This allows improved reasoning about software security.

Vulnerabilities in unused code become active points for exploitable weaknesses [38]. Such vulnerabilities can be potentially exploited via code reuse attacks [39], [40] that can jump to arbitrary locations in the code. Since debloating removes unused code from the target binary, it can reduce the attack surface by pruning code that contains exploitable vulnerabilities. To answer Q5 of our evaluation questions, we identify known common vulnerabilities and exposures (CVEs) that are removed by debloating the programs evaluated with configurations described in the previous section (i.e., Section 5.1). Each CVE is given a severity score according to the Common Vulnerability Scoring System [41] which depends on the ease of the exploit and the impact of the exploit. The severity score ranges from 0 to 10, with 10 being the most severe.

For 9 of the 20 evaluated programs, there have been vulnerabilities reported across different software versions. For 4 of the 9 programs with reported vulnerabilities, we found a total of 5 CVEs that were eliminated by debloating with TRIMMER. For the other programs, most vulnerabilities manifest in core functionality. Hence, these were not eliminated. We describe the CVEs that were removed below:

CVE-2009-4490: CVE-2009-4490 is an execute code vulnerability in *mini_httpd* version 1.19 that occurs when it optionally writes to a log file without properly sanitizing non-printable characters, which might allow attackers to modify or execute arbitrary commands or even overwrite, via an HTTP request containing an escape sequence. Since we specialized *mini_httpd* with logging disabled, the code for logging was fully pruned and hence the vulnerability corresponding to it was eliminated. This CVE is the only vulnerability present in this version of *mini_httpd* and has a severity score of 5.0.

CVE-2010-0001: *gzip* has an integer underflow vulnerabililty with CVE ID 2010-001. The integer underflow occurs in the `unlzw` function in `unlzw.c` before versions 1.4 on 64-bit platforms. This vulnerability allows attackers to cause an application crash (denial of service) or possibly execute arbitrary code via a crafted archive. Since we specialize *gzip* for compression and decompression separately, this vulnerability is eliminated (since the affected function is removed) from the binary specialized for compression. While decompression is a core functionality of *gzip*, it may not be invoked in all usage contexts of the tool.

Similarly, removing decompression functionality from *gzip* removes **CVE-2009-2624** by pruning functions that lead to a

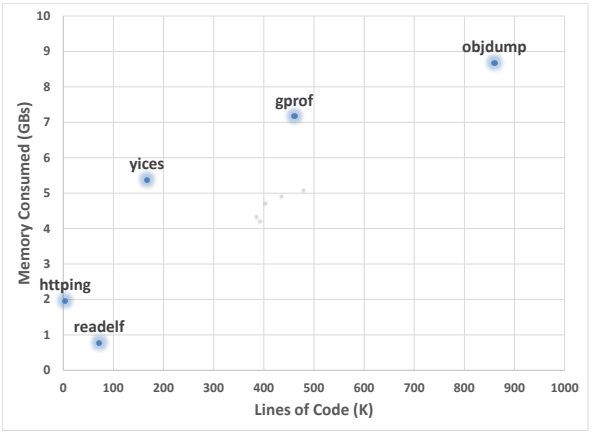


Fig. 11: Peak Memory Consumption by TRIMMER with respect to Lines of Code

denial of service exploit. These two CVEs are the only CVEs present in this version of *gzip* and have a severity score of 6.8.

CVE-2010-0405: Integer overflow in the `BZ2_decompress` function in `decompress.c` in *bzip2* and libbbzip2 before 1.0.6 allows a denial of service attack. This vulnerability is present in the decompression functionality. Similar to *gzip*, we transformed *bzip2* to create specialized binaries for compression and decompression. The version specialized for compression excludes this vulnerability. This CVE is the only vulnerability present in this version of *bzip2* and has severity score of 5.1.

CVE-2017-7209: The `dump_section_as_bytes` function in *readelf-2.28* accesses a NULL pointer while reading section contents, which can lead to a program crash. By specializing *readelf* for a common usage scenario (i.e., the –all option), the affected function is removed. This eliminates the vulnerability. This CVE has severity score of 4.3.

5.6 Scalability of Analyses

We evaluate the peak memory consumption of TRIMMER analyses across the different programs with the goal of finding out if our analyses are feasible in terms of memory usage (Q6 of our evaluation questions). For measuring memory usage, we use *Massif*, a tool in the *valgrind* suite. Figure 11 shows peak memory (Y-axis) consumed by TRIMMER, for the 5 applications with highest memory usage (others omitted for readability) and relates it to the program size in terms of lines of code (X-axis). Unsurprisingly, the memory usage is highest for the largest program, *objdump*, consuming 8.7GB of memory. For *yices* and *gprof*, memory usage is 5.4GB and 7.2GB respectively. For all other programs, memory usage is below 2GB. A major fraction of memory is consumed as part of the SVF code (third party tool) that is invoked for value-flow graph construction used by the configuration annotations transform. This indicates that to scale to larger programs, efficient strategies for value-flow graph construction are important (and are beyond the scope of this work).

Limiting Tracked Values: To control memory usage during constant propagation, we provide an option, *trackedPercent*, to limit the percentage of tracked values. This option is useful for programs that require maintenance of large amounts of shadow memory. Reducing the tracked values limits the number of contexts that need to be maintained simultaneously, thus reducing the memory consumed during constant propagation. To maximize coverage of constant propagation, we select values with the highest number of loads dependent on them. This is done by sorting the list of tainted values by the number of loads dependent on them and considering only the specified percentage of values from the top for tracking.The *trackedPercent* option was not needed in any of our benchmark programs. However, we tested its utility on a larger program (with a baseline binary size of 5.1MB), *Imagemagick* [42]. *Imagemagick* is a tool for creating, editing, and converting digital images. We specialized it for image conversion, but the specialization could not complete on our test machines due to a memory explosion caused by the shadow memory kept during constant propagation. This was fixed by limiting the annotated values (setting *trackedPercent* = 50%). With this, the specialization completed, consuming less than 2 GB of memory in the process. The specialization resulted in a working binary and achieved a code size reduction of 10.72%.

6 THREATS TO VALIDITY

Dependence on Configuration Set: The results obtained in our evaluation regarding code size reduction, attack space reduction and performance are specific to our choice of configuration (per application). With different choices of configurations, the code size reductions and overall achievable benefits (performance, security) are expected to change (may be higher or lower).

Dependence on LLVM version: Our evaluation results for both code size reductions and performance improvements are specific to our choice of LLVM version (LLVM-7). The transformations and their orderings in LLVM optimization levels differ across versions. Due to this, the code size reductions are expected to change when using a different LLVM version with TRIMMER.

7 LIMITATIONS

There are a few limitations of our analysis. We discuss these along with potential proposals for addressing them.

Static Configuration: TRIMMER works under the assumption that the configuration parameters are statically configured once, and the parameters will not change during program execution. If a user desires to change configuration parameters, the application would need to be re-specialized for the updated configuration. TRIMMER also assumes that configuration files are read-only and assumes that there are no intervening file writes. It doesn't reason about scenarios where the file path to one of the configuration files is dynamically loaded and the file is written-to. In our experience, we do not find this to be a common scenario since configuration files are mostly opened in read-only mode.

Process Creation: TRIMMER terminates analysis on process creation calls such as *fork* since it cannot reason about constants across different address spaces.

Lack of Function Devirtualization: TRIMMER currently does not include support for devirtualizing indirect calls (especially common in C++ code). This forces TRIMMER to make conservative assumptions about memory side-effects, therefore limiting constant propagation and dead code elimination. As part of future work, we believe that incorporating support for precise call-graph construction will enable support for a wider range of programs.

8 Related Work

Static Analysis based approaches. Smowton *et al.* [43] proposed LLPE, a LLVM-based partial evaluator that specializes programs with respect to runtime configuration arguments and files. Starting from the program entry-point, it evaluates each instruction and calculates its results if its arguments are known, symbolically executing instructions over a domain of constants, symbolic pointers and symbolic file descriptors. Symbolic execution has known limitations in scaling to large programs and it is not complete when analyzing loops, resulting in the requirement to sometimes keep unspecialized versions of functions. LLPE is primarily focused on improving performance (not code size reduction), and hence this is a reasonable tradeoff for that use case. TRIMMER, in comparison, uses static analysis with a byte-wise arranged concrete memory model to evaluate each instruction. Moreover, in practice, LLPE often requires an interactive process with the developer in the loop guiding the analysis by manually specifying different entry-points (other than 'main' function) for specialization. In comparison, TRIMMER is designed to be fully automated in its analysis. Although LLPE is open-source and publicly available, we did not use it for quantitative analysis since we found that it requires significant manual intervention to correctly specialize a program (i.e., no correctness bugs arising from specialization).

Malecha *et al.* [8] proposed OCCAM, a partial evaluator that specializes programs with respect to constant command-line inputs. OCCAM uses abstract interpretation-based static analysis for dead code elimination. Specifically, OCCAM uses interval analysis on SSA values (in LLVM) and uses these to remove provably dead code branches (and code downstream). OCCAM also uses SeaDSA's [44] (context- and field-sensitive) points-to analysis to compute precise call-graphs. OCCAM's default mode does not provide machinery for constant propagation of memory contents beyond standard LLVM optimizations. Its static analysis does not support specialization with respect to configuration files or custom handling for tracking constants through command line and string processing functions, as TRIMMER does. Our evaluation showed that OCCAM is less effective in achieving code size reductions. Overall, OCCAM and TRIMMER include strategies that are complementary to each other. The precise call-graph construction in OCCAM and support for interval analysis can be combined with TRIMMER to improve code size reductions.

Ghavamnia *et al.* [45] present a novel temporal specialization approach to remove exploitable system calls at program phases beyond which the calls are unlikely to be used. For instance, `execve` calls are commonly exploited in code-reuse attacks, and should be made unavailable to the process beyond programs phases where the call is typically used (usually at initialization). Since many of these calls are made via indirect function invocations with potentially many targets, the authors also propose approaches that remove spurious edges from the derived program call-graph with the goal of improving the analysis precision.

Dynamic analysis based approaches

CHISEL [12] uses learning-based delta debugging for program debloating. It takes as input a program to be debloated and a high-level specification of its functionality to be retained; the rest of the functionality is considered for debloating. The specification is a developer-specified script that includes test cases which should preserve correctness (test case results) after some subset of code (chosen by the tool) is removed. For reasonably sized programs, such a specification mechanism can be burdensome and error-prone since multiple test cases may either need to be carefully selected from an existing test suite or developed from scratch.

Landsborough *et al.* presented two different approaches to code debloating. The first approach uses a dynamic execution-trace of the program to identify unused blocks of code and then replaces these code blocks with NOP instructions. The other approach uses a genetic algorithm to create multiple mutated versions of the program with each mutation removing potentially unused instructions. Each version is then ranked according to a *fitness function* that evaluates if this version passes the user-specified test cases. Like CHISEL, this approach relies entirely on developer-provided test cases and does not provide any guarantees for scenarios/inputs not covered in the test suite.

Qian *et al.* [46] propose a debloating framework, RAZOR, that performs code debloating on the program binary, not requiring source code. It utilizes a set of test cases and control-flow-based heuristics to identify code functions and code blocks that are required to support user-expected functionalities. This approach is useful when program source code is not available. However, a) the lack of source code (or program in IR form) limits potential for program analysis since many high-level program constructs are lost in translation to the program binary, and b) this approach also requires developers to provide extensive test cases.

Kurmus *et al.* [2] use the kernel source and run-time traces to derive a kernel configuration that includes the minimal features that are exercised in the runtime trace collection phase. This kernel configuration is used to recompile the kernel disabling the unused features, resulting in a smaller kernel footprint. TRIMMER provides specialization opportunities at a finer granularity (function, loop, instruction level) compared to what is usually enabled by compile-time configurations. As part of future work, we believe there is potential for using such compile-time configuration specialization approaches with improved constant propagation transforms in TRIMMER to achieve higher code size improvements.

Debloating libraries. Quanch *et al.* [11] use a combination of static and dynamic analyses to eliminate unused functions from libraries at load-time. First, the tool uses static analysis to construct a function-level dependency graph that is appended to the binary header (ELF section). Then a custom system loader uses this dependency graph to prune-out routines (zeroes out corresponding memory pages) that are unused with respect to the external symbol dependencies of the target program.

BlankIt [47] reduces the program attack surface by lazily loading external library functions based on a predictor that predicts the external library calls likely to be invoked at different program points. The predictor used is a decision tree that is embedded in the program binary and has low runtime overhead. The predictor uses the function arguments, and other attributes to predict the external functions callable within an internal function. The approach allows for loading the minimal subset of library functions needed and hence reduces exploitable vulnerabilities.

Other tools debloat library code using static and/or dynamic analyses [48], [49], [50], [51]. Our approach to program debloating is complementary to these techniques and TRIMMER can be potentially combined with these systems. Removing unused application features can create more opportunities for removing unused library interfaces.

Configuration Analysis. Lotrack [52] analyzes application source given configuration options, and uses static taint analysis to identify code fragments that are invoked conditional on the values of these configuration options. The goal of this tool is to

assist developer understanding and is not used for code debloating, though it has potential for being used similar to our custom analysis for identifying configuration-hosting variables. Similarly, Siegmund *et al.* [53] proposed a method for predicting the footprint and memory consumption of a software's non-functional components. This approach only predicts the memory footprint added by different features and is not a tool for code debloating.

9 Conclusion

We introduced TRIMMER, an application specialization tool that debloats unused functionality by specializing programs for user-specified configuration options. We proposed *sparse context-sensitive analysis* for effective constant propagation in reasonable times. *Sparse constant propagation* performs constant propagation for *configuration-hosting* program variables and memory objects that are identified through an analysis pass. Our results demonstrate that our tool effectively optimizes binary sizes for real-world programs. Overall, we observe a mean size reduction of 22.7% across 20 evaluated programs.

10 Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant ACI-1440800 and the Office of Naval Research (ONR) under Contract N68335-17-C-0558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

References

[1] 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 *3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[2] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring," in *20th Network and Distributed System Security Symposium*, 2013.

[3] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *FSE/SDP Workshop on Future of software engineering research*, 2010.

[4] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, "Software bloat and wasted joules: Is modularity a hurdle to green software?" *Computer*, vol. 44, no. 9, 2011.

[5] G. Muller, R. Marlet, E.-N. Volanschi, C. Consel, C. Pu, and A. Goel, "Fast, optimized sun rpc using automatic program specialization," in *18th International Conference on Distributed Computing Systems*, 1998.

[6] S. Bhatia, C. Consel, A.-F. Le Meur, and C. Pu, "Automatic specialization of protocol stacks in operating system kernels," in *29th IEEE International Conference on Local Computer Networks*, 2004.

[7] S. Bhatia, C. Consel, and C. Pu, "Remote specialization for efficient embedded operating systems," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 4, 2008.

[8] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing," in *30th ACM Symposium on Applied Computing*, 2015.

[9] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha, "New directions for container debloating," in *2nd Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[10] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *11th Joint Meeting on Foundations of Software Engineering*, 2017.

[11] A. Quach, A. Prakash, and L. Yan, "Debloating software through piecewise compilation and loading," in *27th USENIX Security Symposium*, 2018.

[12] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[13] 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*, 2019.

[14] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[15] C. Hibbs, S. Jewett, and M. Sullivan, *The art of lean software development: a practical and incremental approach*. O'Reilly Media, Inc., 2009.

[16] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *35th Annual Design Automation Conference*, 1998.

[17] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *25th International Conference on Compiler Construction*, 2016.

[18] O. Danvy, "Type-directed partial evaluation," in *Partial Evaluation*, 1999.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

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

[21] "LoopRotate] Add support for rotating loops with switch exit," https://reviews.llvm.org/D72420.

[22] "mini_httpd," https://acme.com/software/mini_httpd/.

[23] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *6th Workshop on Networking Meets Databases*, 2011.

[24] S. Choudhuri and R. N. Mahapatra, "Energy characterization of filesystems for diskless embedded systems," in *41st Design Automation Conference*, 2004.

[25] "sans," https://github.com/puxxustc/sans.

[26] "totd," http://www.dillema.net/software/totd.html.

[27] J. Poskanzer, "thttpd - tiny/turbo/throttling httpserver," https://acme.com/software/thttpd/.

[28] "bzip2," https://linux.die.net/man/1/bzip2.

[29] T. Lu, W. Xia, X. Zou, and Q. Xia, "Adaptively compressing iot data on the resource-constrained edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing*, 2020.

[30] [Online]. Available: https://linux.die.net/man/1/gzip

[31] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur *et al.*, "Memcached design on high performance rdma capable interconnects," in *International Conference on Parallel Processing*. IEEE, 2011, pp. 743–752.

[32] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*. Springer, 2014, pp. 737–744.

[33] E. Clarke, M. Talupur, H. Veith, and D. Wang, "Sat based predicate abstraction for hardware verification," in *Theory and Applications of Satisfiability Testing*. Springer, 2003, pp. 78–92.

[34] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[35] "Everthing curl." [Online]. Available: https://everything.curl.dev/

[36] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[37] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[38] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments," in *2nd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[39] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *14th ACM Conference on Computer and Communications Security*, 2007.

[40] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[41] "Common vulnerability scoring system sig." [Online]. Available: https://www.first.org/cvss/

[42] "imagemagick," https://imagemagick.org/index.php.

[43] C. Smowton and S. Hand, "make world," in *13th USENIX Workshop on Hot Topics in Operating Systems*, 2011.

[44] "SeaDSA," https://github.com/seahorn/sea-dsa.

[45] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium*, 2020.

[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*, 2019.
[47] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you don't," in *41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
[48] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," *Black Hat USA*, 2015.
[49] I. Agadakos, D. Jin, D. Williams-King, V. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *35th Annual Computer Security Applications Conference*, 2019.
[50] A. Ziegler, J. Geus, B. Heinloth, T. HÖnig, and D. Lohmann, "Honey, i shrunk the elfs: Lightweight binary tailoring of shared libraries," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, 2019.
[51] N. Davidsson, A. Pawlowski, and T. Holz, "Towards automated application-specific software stacks," in *24th European Symposium on Research in Computer Security*, 2019.
[52] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, 2017.
[53] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption," *Information and Software Technology*, vol. 55, no. 3, 2013.

Aatira Anum Ahmad has completed her Master's and is pursuing a Doctorate in Computer Science, both at Lahore University of Management Sciences. Her research interests are compilers and program analysis.

Abdul Rafae Noor is currently a Computer Science Ph.D. student in the LLVM group at the University of Illinois, Urbana-Champaign. His research interests include generating and optimizing hardware-agnostic code for heterogeneous parallel systems.

Hashim Sharif completed his Ph.D. and is now a Postdoctoral Fellow, both at the University of Illinois, Urbana-Champaign. His research experiences include internships at SRI International and Argonne National Laboratory. Hashim is enthusiastic about building compiler infrastructure that improves performance and reduces energy usage on resource-constrained systems. His research interests include compilers, static analysis, approximate computing, and deep learning.

Usama Hameed received his Bachelor's in Computer Science from Lahore University of Management Sciences. He is pursuing his Ph.D. at University of California, Los Angeles. His research interests include Compilers, Software Engineering, and Static Analysis.

Shoaib Asif is a Ph.D. student at University of Texas, Austin. He is involved in two research projects that focus on debloating software. He is interested in using program analysis to solve security challenges.

Mubashir Anwar received his undergraduate degree at Lahore University of Management Sciences. He has joined the Ph.D. program at the University of Illinois, Urbana-Champaign. His research interests include static analysis, software-defined networking, and edge-computing.

Ashish Gehani is a Principal Computer Scientist at SRI. His research interests are data provenance and security. He holds a Ph.D. in Computer Science from Duke University and a B.S. in Mathematics from the University of Chicago.

Fareed Zaffar is an Associate Professor of Computer Science at Lahore University of Management Sciences. His primary research interests are in the areas of security, privacy, and internet measurement. His recent work has focused on social networks, online fraud, and cybercrime. He has also done extensive work in enabling public sector reform through the use of information and communication technologies. His work has been supported by HEC, Facebook, USAID, NIH Pakistan, GIZ, UNODC, European Union, USIP, Adam Smith International, and UNICEF, among others.

Junaid Haroon Siddiqui is an Associate Professor of Computer Science at Lahore University of Management Sciences. Previously, he received Ph.D. in Computer Science from University of Texas, Austin, and M.S. and B.S. in Computer Science from FAST-NUCES. His research interests include program analysis using static and dynamic techniques in automatic software test generation, parallel and incremental techniques in scaling algorithms for multicore processors, and the intersection of these domains.