Trimmer: Context-Specific Code Reduction

Aatira Anum Ahmad
17030061@lums.edu.pk
LUMS
Lahore, Pakistan

Mubashir Anwar
manwar@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Hashim Sharif
hsharif3@illinois.edu
University of Illinois at Urbana-Champaign
Champaign, IL, USA

Ashish Gehani
ashish.gehani@sri.com
SRI International
Menlo Park, CA, USA

Fareed Zaffar
fareed.zaffar@lums.edu.pk
LUMS
Lahore, Pakistan

ABSTRACT

We present Trimmer, a state-of-the-art tool for reducing code size. Trimmer reduces code sizes by specializing programs with respect to constant inputs provided by developers. The static data can be provided as command-line options or through configuration files. The constants define the features that must be retained, which in turn determine the features that are *unused* in a specific deployment (and can therefore be removed). Trimmer includes sophisticated compiler transformations for input specialization, supports precise yet efficient context-sensitive inter-procedural constant propagation, and introduces a custom loop unroller. Trimmer is easy-to-use and extensively parameterized. We discuss how Trimmer can be configured by developers to explicitly trade analysis precision and specialization time. We also provide a high-level description of Trimmer's static analysis passes. The source code is publicly available at: https://github.com/ashish-gehani/Trimmer. A video demonstration can be found here: https://youtu.be/6pAuJ68INnI.

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Static analysis, Code debloating, Program specialization, LLVM

ACM Reference Format:
Aatira Anum Ahmad, Mubashir Anwar, Hashim Sharif, Ashish Gehani, and Fareed Zaffar. 2022. Trimmer: Context-Specific Code Reduction. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559529

1 INTRODUCTION

Modern applications contain much functionality of which only a limited subset of features are used in any particular deployment. These unused features (referred to as code bloat) decrease an application's performance, increase resource consumption, and raise the probability of latent exploitable vulnerabilities [8, 9]. They impose unnecessary load and security risks on devices, such as embedded platforms with limited memory and power.

To counter code bloat, multiple prior studies [1, 3–5, 7, 10, 12, 13] have proposed static (compile-time) and dynamic (run-time) techniques. Dynamic techniques [3, 7, 11] are slow and require a set of detailed test cases that correspond to functionality that is intended to be preserved. Some tools require users to understand the details of the application code [7]. In contrast to some static analysis-based tools, such as LLPE [13], that require significant manual intervention, Trimmer is completely automated.

We describe Trimmer, an automated static analysis-based tool that specializes a program for a specific deployment, removing code that is unused in the target context. The usage context is defined by the user through command-line inputs and application-specific configuration files. Trimmer is LLVM-based infrastructure that uses custom compiler analysis for input specialization, constant propagation, and loop unrolling, with the goal of facilitating dead code elimination. Trimmer takes as input a manifest file that includes (i) constant command-line inputs to use for specialization , (ii) paths to application-specific configuration files (if any) that include constant configuration settings to use during specialization, and (iii) the path to an LLVM bitcode module for the input program. As output, it generates an executable binary specialized with respect to the given static configuration settings. Trimmer is carefully designed to be *easy-to-use* - the manifest file does not require developer expertise and only requires high-level configuration parameters.

2 USAGE

Trimmer is invoked through a Python driver script, shown below:

```
python $TRIMMER_HOME/tool/trimmer.py [manifest-file]
   [working-dir] [options]
```

The arguments to the script are: (i) the path to the *manifest file* that describes the specialization context for a target application, (ii) the path to the working directory where the intermediate and final specialized files will be stored, and (iii) the list of options passed to Trimmer. If options are not explicitly provided, Trimmer uses a default set of values. The manifest file and the configurable Trimmer options are described next.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559529

```
{
    "main": "wget.bc",
    "name": "wget",
    "binary": "wget_specialized",
    "args": ["--config=wgetrc","_" ],
    "modules": [],
    "native_libs":[],
    "ldflags": ["-luuid","-lgnutls",
    "-lpcre", "-lnettle", "-lidn",
    "-lz", "-lpthread"],
    "config_files": ["wgetrc"]
}
```

Figure 1: Example Trimmer Manifest File

2.1 Manifest File

The *manifest file* contains the following key-value pairs in JSON format:

- **main:** path to the input LLVM bitcode module.
- **binary:** name of the specialized binary that is emitted by Trimmer.
- **name:** name of the application, which is used as the first `argv` parameter.
- **args:** the list of arguments to use for specialization. Static arguments used for specialization are specified as constant strings, while dynamic arguments are specified with the *underscore* symbol "_". Dynamic arguments can be passed as runtime values to the specialized binary.
- **modules:** (optional) a list of LLVM bitcode module dependencies to be linked (e.g., static libraries).
- **config_files:** (optional) a list of application-specific configuration file(s) to use for specialization.
- **native_libs:** (optional) a list of external libraries or object files to be linked against the LLVM bitcode modules (e.g., precompiled headers)
- **ldflags:** (optional) a list of linker flags.

Figure 1 shows the manifest file we use for *wget* [6], a tool used for scripted retrieval of content from web servers. The `args` parameter specifies the configuration arguments to use for specialization - this includes a path to the configuration file *wgetrc* and _ (underscore) as the dynamic argument for the target URL.

2.2 Trimmer Options

Trimmer provides options to customize the specialization for every run.

Some important configurable options are:

Context Type: Trimmer supports both *context-insensitive* (one clone per function) and *context-sensitive* (one clone per calling context) constant propagation. The default is a *sparse context-sensitive* mode in which expensive context-sensitive constant propagation is limited to the program slice that is *tainted* by *configuration hosting variables* [2]. This program slice is likely to include opportunities for debloating code with respect to static configuration values.

Clone Limit: By default, Trimmer clones a specific function for each unique context (i.e. set of input arguments) to create specialized function variant. The maximum number of clones per function is limited by using the *restrictLimit* option. This option is useful when dealing with recursion. It is also valuable when excessive cloning increases code size.

Tracked Percent: In sparse context-sensitive mode, the *trackedPercent* option limits the number of configuration variables that are analyzed in the constant propagation pass. The rationale for this option is discussed in Section 3.1.

Use Globals: By default, all global variables are treated as non-constant for specialization purposes to avoid excessive function cloning. Since configuration values are rarely ever stored in globals, Trimmer does not create specialized clones for functions by default. This saves analysis time as it reduces the variables and functions that need to be analyzed. For cases where configuration variables are stored in globals, the user can enable the *Use Globals* option.

3 WORKFLOW / IMPLEMENTATION

Figure 2 shows the workflow of the Trimmer tool. Trimmer takes as input a program compiled to a single LLVM bitcode module (which can be achieved using *llvm-link*) and a user-configurable manifest file (shown in Figure 1) that includes the constant parameters to be used for *program specialization*. Trimmer includes five compiler transforms that synergistically propagate constants throughout the program control flow graph. These include passes for configuration annotations, command-line input specialization, loop unrolling, file input specialization, and custom constant propagation. Constant propagation facilitates the folding of conditional expressions. For example, `if` statements dependent on values that can be determined are candidates for simplification. This in turn facilitates the removal of basic blocks that are no longer reachable (and are therefore *dead code*).

Configuration annotations, the first pass in the Trimmer workflow, identifies program variables that *may* host configuration values. The goal of this pass is to reduce the number of program variables that need to be analyzed by the constant propagation pass. Trimmer includes two different passes for incorporating user-specified configuration constants in the program control flow graph: (a) *command-line input specialization*, and (b) *file input specialization*. Trimmer provides a novel *sparse context-sensitive constant propagation* transform, and a *custom loop unrolling* pass that uses constant propagation results to provide more precise loop unrolling. Since the constant propagation, file input specialization, and loop unrolling passes have an iterative interaction with each other (e.g., constant propagation facilitates loop unrolling and vice versa), these are developed as a single compiler transform for ease and efficiency of implementation. However, since these are conceptually different static analyses, we discuss them separately. After these passes, the standard LLVM compiler optimizations are invoked - these include passes for global dead code elimination. The optimized LLVM modules are then compiled into a native binary.

The algorithmic details of these transforms are described in the Trimmer journal paper [2]. We summarize some of the key implementation details below.

3.1 Identifying Configuration-hosting Variables

The *configuration annotations* pass performs taint analysis to identify variables that may potentially be modified by one or more

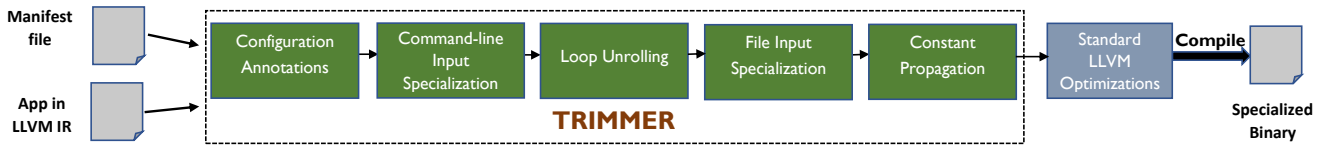

Figure 2: TRIMMER's workflow

configuration values used for specialization. The analysis involves: (1) a value flow graph (VFG) construction step that builds an in-memory representation that captures both the control and data dependencies of the program, followed by (2) a traversal over this VFG to identify and mark tainted variables. For value flow graph construction, we use SVF [14], a third-party tool that provides various analyses including pointer analysis, call-graph analysis, mod-ref analysis, and memory-SSA construction. We also provide users with the ability to control the percentage of tainted variables through an option, *trackedPercent*, that takes a value between 0 (no variables tracked) and 100 (all variables tracked). The tainted variables are sorted by the number of `load` instructions of the target variable. Variables with more loads are given higher priority for analysis. For example, a value of *trackedPercent = 5* means that of the variables with highest number of loads, only the top 5% are tracked for the constant propagation analysis.

3.2 Input Specialization

TRIMMER provides an intuitive and easy-to-use interface for describing the intended deployment context. Users specify this context through command-line inputs and configuration files.

3.2.1 Command-line input specialization. This pass specializes the program's `main` function with respect to static values in the `argv` array specified in the manifest. The pass replaces loads (at the LLVM level) from `argv` elements with corresponding constant values. These constants are then propagated throughout the program's control flow graph by our custom constant propagation pass.

3.2.2 File input specialization. This pass helps developers turn application-specific configuration files into the target of specialization. Developers can specify both command-line inputs and configuration files, or either of the two based on how a particular program uses these parameters. The path to the configuration file is specified in the manifest.

The file input specialization pass supports file `open`, `read`, `seek`, and `close` operations. Write operations are not handled (beyond creating an exception) since this violates the assumption of constant configuration input. The pass performs a flow-sensitive analysis on the file operations. It maintains and updates a *file context* for each open file. This file context is an in-memory data-structure used to track state such as file descriptors and their currently associated offsets. The pass updates the contexts appropriately on each `open`, `read`, `seek`, and `close` operation.

3.3 Loop Unrolling

Loop unrolling is necessary to facilitate constant propagation. The default LLVM loop unrolling pass does not unroll loops with non-constant trip count - i.e. an unknown number of iterations. This leads to missed opportunities for specialization. Another limitation is that it uses a static heuristic that estimates the dynamic cost of running the instructions in unrolled form and only unrolls loops that reduce this cost - e.g., when loads in a loop can be folded to constants. This heuristic is limited in that it cannot accurately predict if loop unrolling will eventually facilitate constant propagation downstream in the program's control flow graph. TRIMMER's loop unroller addresses both these issues with *speculative unrolling* - the loop is peeled/unrolled for a certain number of iterations (set to 20 in our evaluation) and constant propagation is applied to the unrolled loop body. This results in the constant folding of loop iterators (or loop index variables) that provides further opportunities for the constant folding of program expressions in the loop body. If the loop is fully unrolled after constant propagation *and* the unrolled loop has constants folded, the loop is left unrolled. Otherwise, the loop is re-rolled.

3.4 Constant Propagation

LLVM's standard constant propagation pass only performs constant folding on *virtual registers*. It does not reason about variables in memory. Reasoning about virtual registers is easier than memory since virtual registers are SSA values with explicit use-def chains. Variables in memory can be aliased through pointers. (They can be accessed in LLVM via the `getElementPointer` instruction.) Hence, reasoning about constants in memory requires a flow-sensitive points-to analysis combined with concrete memory tracking (that follows the locations of constants). We developed a custom inter-procedural constant propagation LLVM pass that reasons about static data in memory using flow-sensitive and context-sensitive *concrete memory tracking*. It maintains a *shadow memory state* for heap and stack allocations (effected by `malloc` and `alloca`, respectively, in bitcode) that is updated when it encounters `load` and `store` instructions. Context-sensitive analyses are known to face scaling challenges. Hence, we introduce a *sparse context-sensitive* analysis that limits concrete memory tracking to just the configuration-hosting variables. TRIMMER also supports a *context-insensitive* analysis mode - it may be less effective at specialization than sparse context-sensitive analysis but will have faster running time. If a pointer escapes a module (in the case of an external function call, for example), conservative assumptions are made about memory side-effects; any corresponding memory allocations are marked as non-constant.

3.5 Example Code

We walk through the TRIMMER transformations passes with a simple C example in Figure 3. We show C code for ease of explanation. The actual transformations are performed on LLVM IR (intermediate representation). The code snippet prompts for a configuration

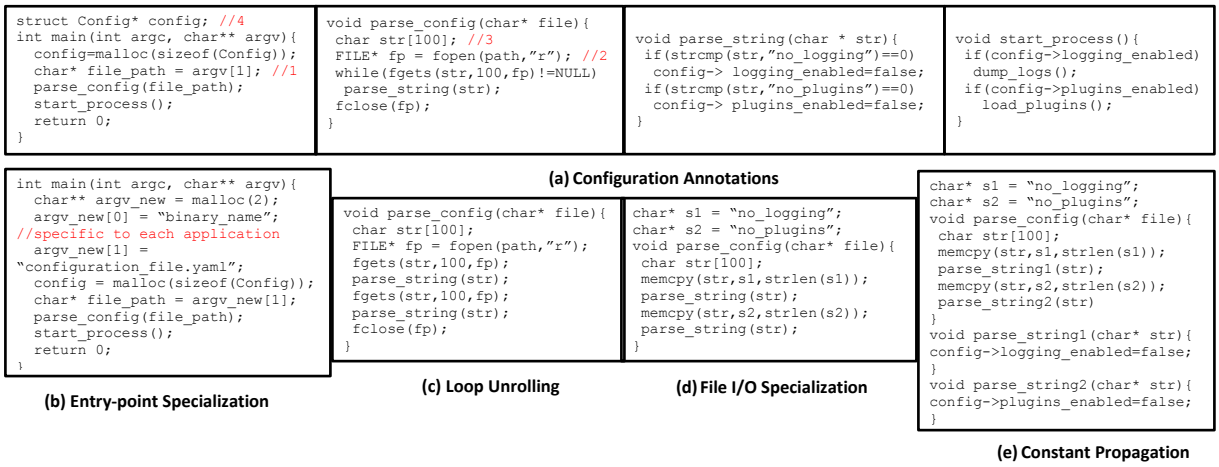

Figure 3: (a) shows a C example code with 4 functions. The variables tainted by configuration annotations pass are numbered. (b) shows main function after command-line input specialization. (c) shows *parse_config* function after loop unrolling. (d) shows *parse_config* function after file I/O specialization. (e) shows *parse_config* and *parse_string* function after context-sensitive constant propagation. Function *parse_string* is cloned twice, one for each callsite.

file name from the command-line, reads the configuration file, and populates the configuration variable, `Config`, with configuration parameters read from the file. Figure 3(a) shows the output of the configuration annotation pass with four variables marked as configuration hosting variables, since they are dependent on the user input passed in `argv[1]`. Figure 3(b) shows the result of running the command-line input specialization pass, with Trimmer creating a new array variable to host the static values of `argv`. The uses of `argv` are appropriately replaced. Figure 3(c) shows the workings of the loop unrolling pass that fully unrolls the loop in the `parse_config` function. Figure 3(d) shows how the file I/O specialization pass replaces file read calls with `memcpy` calls from global constant strings. These global constant strings are created to host the statically determined constant file data. Figure 3(e) shows the code after the constant propagation pass is applied to the code in (d). For the original function, `parse_string`, the constant propagation pass creates two specialized clones, `parse_string1` and `parse_string2`. Each cloned version is specialized for a particular constant argument `str` (corresponding to a parameter read from the configuration file). The conditional (`if`) statements predicated on `strcmp` calls in `parse_string1` and `parse_string2` are statically evaluated with respect to the constant strings. This constant folding reduces the configuration parameters `logging_enabled` and `plugins_enabled` to the constant `False` (assuming they were `False` in the example configuration). This later allows for dead code removal of the functions `load_plugins` and `dump_logs`.

4 EVALUATION

Benchmarks and Setup: We evaluate Trimmer on 20 commonly-used Linux utilities. We selected a diverse set of applications including web servers, compression tools, networking utilities, and an SMT solver (`yices`). To construct realistic specialization scenarios, we chose configurations that represent the core functionality of the application while leaving out secondary (less used) features.

We conducted experiments using Ubuntu 16.04 as the operating system, running on a machine with a 1.8GHz frequency Intel Core i7 (8th generation) processor configured with 16GB of RAM. The baseline versions of programs used for size comparisons were compiled with the *-Os* option (to optimize for size). For performance comparisons, the baselines were compiled with *-O3*. We use application-specific benchmarks when available for performance measurements. Otherwise, we measure execution time on sample inputs.

Code Size Impact: Trimmer provides a mean and maximum code size reduction of 20.4% and 61.2%, respectively.

Performance Impact: The largest speedup observed was 53% (aircrack-ng). For 5 of the applications(aircrack-ng, netperf, yices, objdump, memcached), a performance improvement of 5% or more was observed.

Security Impact: Trimmer removed 5 known vulnerabilities in 4 of the evaluated programs.

The detailed evaluation results are discussed in the journal publication [2].

5 SCOPE AND LIMITATIONS

Trimmer supports programs that can be compiled to LLVM IR. It is most likely to provide size reductions in programs that include multiple features, only a subset of which are used in a particular deployment. Currently, Trimmer does not support function devirtualization (for resolving indirect calls). Due to this limitation, it must make conservative assumptions about memory side-effects. This potentially limits the code size reduction from specialization of programs with polymorphic functions (as may be present in C++ applications, for example).

ACKNOWLEDGMENTS

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 Contracts N68335-17-C-0558 and

N00014-18-1-2660. 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] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-Scale Debloating of Binary Shared Libraries. *Digital Threats* 1, 4 (2020).

[2] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed Zaffar. 2021. TRIMMER: An Automated System for Configuration-based Software Debloating. *IEEE Transactions on Software Engineering* (2021).

[3] Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code specialization through Dynamic Feature Observation. *11th ACM Conference on Data and Application Security and Privacy* (2021).

[4] Michael D Brown and Santosh Pande. 2019. Is less really more? Towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test*.

[5] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. *JShrink: In-Depth Investigation into Debloating Modern Java Applications.* ACM.

[6] GNU. 2020. GNU wget. "https://www.gnu.org/software/wget/".

[7] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *25th ACM Conference on Computer and Communications Security*.

[8] Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. 2018. BinRec: Attack surface reduction through dynamic binary recovery. In *3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*.

[9] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack surface metrics and automated compile-time OS kernel tailoring. In *20th Network and Distributed System Security Symposium*.

[10] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *30th ACM Symposium on Applied Computing*.

[11] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium*.

[12] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium*.

[13] Christopher Smowton. 2011. Make world. In *13th Workshop on Hot Topics in Operating Systems*.

[14] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *25th International Conference on Compiler Construction*.