TRIMMER: Application Specialization for Code Debloating

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

Muhammad Abubakar
Department of Computer Science
Lahore University of Management Sciences
Lahore, Pakistan
bakar95@gmail.com

Ashish Gehani
Computer Science Laboratory
SRI International
Menlo Park, USA
ashish.gehani@sri.com

Fareed Zaffar
Department of Computer Science
Lahore University of Management Sciences
Lahore, Pakistan
fareed.zaffar@lums.edu.pk

ABSTRACT

With the proliferation of new hardware architectures and ever-evolving user requirements, the software stack is becoming increasingly bloated. In practice, only a limited subset of the supported functionality is utilized in a particular usage context, thereby presenting an opportunity to eliminate unused features. In the past, program specialization has been proposed as a mechanism for enabling automatic software debloating. In this work, we show how existing program specialization techniques lack the analyses required for providing code simplification for real-world programs. We present an approach that uses stronger analysis techniques to take advantage of constant configuration data, thereby enabling more effective debloating. We developed Trimmer, an application specialization tool that leverages user-provided configuration data to specialize an application to its deployment context. The specialization process attempts to eliminate the application functionality that is unused in the user-defined context. Our evaluation demonstrates Trimmer can effectively reduce code bloat. For 13 applications spanning various domains, we observe a mean binary size reduction of 21% and a maximum reduction of 75%. We also show specialization reduces the surface for code-reuse attacks by reducing the number of exploitable gadgets. For the evaluated programs, we observe a 20% mean reduction in the total gadget count and a maximum reduction of 87%.

CCS CONCEPTS

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

KEYWORDS

Static analysis, Code debloating, Program specialization, LLVM

ACM Reference Format:
Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238160

1 INTRODUCTION

As hardware architectures and user requirements continue to evolve, software platforms and applications are becoming increasingly complex [33]. With such software growth, it is not surprising to see applications becoming subject to feature creep. As a variant of the 80/20 rule, it has been argued that 80% of the users only utilize 20% of the functionality with each user requiring a distinct set of features [24]. With most users demanding only a subset of the complete functionality, feature extensions often come at the cost of software bloat. Such code bloat may negatively impact performance, while also potentially broadening the attack surface of the application [7, 40, 52]. Moreover, software bloat is a first-class concern in embedded systems hosted on resource-constrained devices with limited memory and secondary storage capabilities [36].

Protocol stacks for embedded systems often need to be highly customized in view of their target applications. This requires developers to manually identify and eliminate unused code from the stack [35]. Such manual stripping of unnecessary features can be a time-consuming and error-prone process. Alternatively, program specialization techniques have been acknowledged as a useful mechanism for automatic customization of software [41].

Conceptually, a specializer takes a target application and a specialization context as input, and produces a version of the application specialized for the provided context. The specialization context is composed of known data values and configuration parameters available at compile time. Prior work has leveraged partial evaluation as a technique for program specialization [6, 16]. Partial evaluation is an optimization technique that precomputes program expressions in terms of the known static input, thereby generating a specialized binary [28].

In this work, we show how existing partial evaluation mechanisms are insufficient for providing useful program specialization for real-world programs. Specifically, the current approaches for automated specialization are not equipped with the analyses that cater to the diverse programming patterns employed for reading and parsing user configuration. Moreover, the absence of propagating configuration invariants, coupled with the conservative nature of existing optimizations in production compilers [17], limits opportunities for constant folding and dead code elimination.

Accordingly, we have developed Trimmer, a tool that enables configuration-based software slimming. Trimmer specializes a target program with respect to a user-defined configuration. The provided configuration specifies the usage context of the application in the deployment of interest, and is leveraged to prune unused functionality from the application. Our tool uses stronger techniques to incorporate configuration information, and includes compiler transformations tailored to take advantage of the known program constants, thereby enabling effective debloating. Instead of relying on heroic compiler analysis, we develop the necessary transforms that enable effective propagation of configuration invariants. In particular, we build an interprocedural constant propagation transformation that is more precise, and hence facilitates the pruning of program paths that are unreachable in the context of the user configuration. The unreachable paths are pruned as a result of statically evaluating branch conditions by leveraging the provably constant data. Moreover, this potentially allows for pruning functions that are invoked within the eliminated program path.

Additionally, we show existing transformations such as loop unrolling can be applied more usefully in light of the known static input. Our experiments show, by applying these additional transforms, we achieve more precise constant propagation, as a result enabling aggressive elimination of unused program code. The compiler transforms included in the tool are developed using the LLVM compiler framework [34].

Our specific contributions can be summarized as follows:

- We develop an interprocedural constant propagation transformation that propagates the static program configuration more precisely, and hence facilitates the pruning of unreachable code paths.
- We develop a loop unrolling transformation to more aggressively unroll loops. In particular, we leverage the improved constant folding capabilities to better guide the cost models that drive the loop unrolling transform. For the majority of programs, full loop unrolling for certain loops (particularly input parsing loops) is a prerequisite for any useful constant propagation.
- We present the results of an evaluation of our tool. Our experiments demonstrate:
  Reduced code size: For a set of 13 commonly used Linux applications, we observe a geometric mean binary size reduction of 21.1% and a maximum reduction of 75.4%.
  Reduced attack surface: Code-reuse attacks use instruction sequences (called gadgets) from executable code sections as the attack payload. We show specialization introduces software diversity and reduces the number of exploitable gadgets. In our evaluation, we observe specialization reduces the gadgets by 20.1% (on average). Moreover, for all our benchmarks, applying specialization eliminates all surviving gadgets (at known byte locations).

2 MOTIVATION

We discuss our motivations for application specialization.

Eliminating auxiliary program features: Prior studies have shown a significant fraction of program features are rarely used [24]. We view this as an opportunity to eliminate the less commonly used features by specializing applications to their core functionality. In our experiments, we aim to select configurations that represent the core functionality of the target applications while leaving out the auxiliary features.

Reduced attack surface: Modern operating systems protect against code injection attacks by preventing pages from being simultaneously executable and writable [49]. Code reuse attacks circumvent these restrictions by using instruction sequences in the executable sections as the attack payload itself [10, 46]. These code snippets, called gadgets, end in a return or jump instruction, thus allowing an attacker to chain the execution of multiple gadgets [26]. As specialization eliminates unused program features, it reduces the attack surface of the program by reducing gadgets. In our results, we show a reduction in binary size correspondingly reduces the set of program gadgets an attacker can exploit. To conduct a code reuse attack, an attacker must have knowledge of the gadget locations in the executable [32]. We show specialization introduces software diversity, making it harder for attackers to gain insights about the gadget locations in the target binary.

3 TRIMMER

In this section, we describe the workflow for our configuration-based debloating tool Trimmer. For our implementation, we selected the LLVM framework [34] since it provides frontends for popular programming languages including C/C++, and supports a wealth of compiler analyses and optimization passes that make it suitable for developing new compiler transforms. Trimmer is intended to be used with software that is compiled to LLVM bitcode modules - bitstream container format for encoding LLVM Intermediate Representation (IR). Figure 1 shows the workflow for the proposed system.

The input to the tool is a manifest file that includes the user-defined static configuration, and the path to a whole-program LLVM bitcode file. The user-defined static configuration characterizes the usage context of the application for a given deployment. Trimmer specializes the target application modules with respect to this usage context. The whole-program bitcode file consists of application code that is compiled and linked into a single LLVM module.

Trimmer is composed of three major compiler transforms namely input specialization, loop unrolling, and constant propagation. The first phase of the tool workflow includes input specialization. Input specialization propagates the static configuration data into the program via the program entry point. This, in turn, facilitates simplifying program expressions dependent on these configuration constants. Input specialization is described in more detail in Section 3.1. Next in the pipeline is a loop unrolling pass that aggressively unrolls loops to aid later optimization passes. We observe that improved loop unrolling is necessary to facilitate effective interprocedural constant propagation. The transform is discussed in Section 3.2.

Once the configuration parameters are incorporated as program invariants, they can be aggressively propagated throughout the

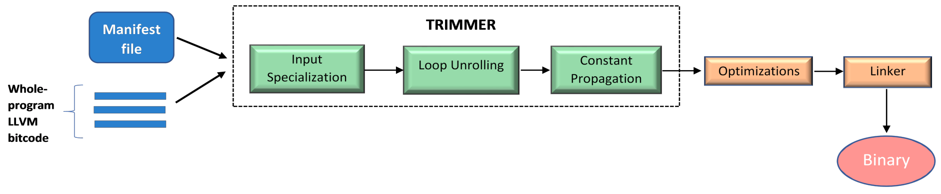

Figure 1: Overview of the tool workflow. Inputs to the tool include a) a whole-program LLVM bitcode file, and b) a manifest file. The user-defined configuration specified in the manifest is leveraged to generate a specialized binary.

program. Production compilers such as LLVM provide optimization passes for constant propagation that enable this goal. However, as LLVM optimizations tend to be conservative for purposes of correctness and efficiency [2, 17], this leads to missed opportunities for optimizing the code more aggressively in light of the statically known values. Accordingly, we develop a more sophisticated inter-procedural constant propagation transform that better propagates the static configuration data throughout the program callgraph, thereby enabling code simplification. The implementation of the constant propagation optimization is described in more detail in Section 3.3. As configuration data is incorporated and propagated, existing compiler optimizations can be invoked later in the pipeline to further attempt code simplification. Finally, the linker takes as input the specialized bitcode and the linker flags (also read from the manifest file), and generates a specialized binary executable.

3.1 Input Specialization

Applications parse user-provided configuration in a variety of different ways. Commonly, configuration is supplied as program arguments and configuration files. While our tool includes transformations for incorporating command-line arguments, supporting file-based configuration is not within the scope of the current work. In practice, we observe a majority of applications include a wealth of configurable command-line flags that can be exploited for specialization.

We describe our input specialization transform that leverages partial evaluation concepts [18, 28]. The goal of this transform is to incorporate user-defined constant inputs directly into the program, therefore specializing the application for a specific usage context. Later passes such as loop unrolling and constant propagation can leverage these constants to simplify program expressions, consequently facilitating code debloating. Specifically, the transform specializes the program entry point for the user-provided input arguments. These static program arguments are provided as part of the manifest file that is input to the specializer.

Algorithm 1 shows the algorithm for the input specialization transform. The pass begins by reading the input arguments from the manifest file. Then, for each constant argument specified in the manifest, all original uses of the argument are substituted by the known constant value. If a target argument is explicitly annotated as dynamic input (as part of the manifest), the original uses remain unsubstituted. This is particularly useful, as it allows users to specialize programs on a partially-specified set of constant arguments.

As a practical example of input specialization, consider a network monitoring tool that takes as input a particular network interface.

Algorithm 1: Algorithm for Input Specialization

```
Function inputSpecialization(manifestFile, programEntry)
    inputArgs ← readArgs(manifestFile);
    programArgs ← getProgramArgs(programEntry);
    argIndex ← 0;
    foreach argValue in inputArgs do
        if argValue is constant then
            replaceArgUses(argValue, programArgs[argIndex]);
        end
        argIndex ← argIndex + 1;
    end
```

While the application may support multiple network interfaces, a usage scenario may only require monitoring a specific interface (e.g., Ethernet). Specializing the application for a particular interface allows for pruning unused functionality, thus reducing code size.

3.2 Loop Unrolling

In this section, we introduce our loop unrolling transformation. In the context of program specialization, we observe that loop unrolling is a necessary optimization. Unrolling is particularly important for input parsing loops that extract the user-provided configuration and store these values as program variables. Since unrolling facilitates the folding of program expressions inside the loop body, it supports constant propagation of the static configuration.

While LLVM includes a loop unrolling pass in the optimizations suite, it is not entirely suited for our use case. Since the LLVM unrolling pass is driven by conservative heuristics to estimate the profitability of the loop unroll, it very often misses opportunities for simplifying expressions in the loop body. Accordingly, we develop a loop unrolling transform that leverages our improved constant folding transform to more precisely determine the profitability of unrolling a target loop. Specifically, the transform aggressively unrolls loops and thereafter applies a cost model on the unrolled loop. In the context of a specific loop, the cost model determines the usefulness of the transform by identifying loop-based operations that are simplified/folded as a result of the unrolling. Since the transform relies on constant folding to drive the cost model, our tool bundles the loop unrolling transform and the constant propagation transform as one unified compiler pass. Constant propagation is described in detail in Section 3.3.

Algorithm 2 illustrates the algorithm for the unrolling transformation. The `tryToUnrollLoop` procedure attempts to fully unroll the provided target loop. First, the call to `getLoopTripCount` extracts the trip count for the loop. If the loop trip count can be statically determined as a constant value, the loop is fully unrolled.

Algorithm 2: Algorithm for Loop Unrolling

```
Function tryToUnrollLoop(loop)
    tripCount ← getLoopTripCount(loop);
    if isConstant(tripCount) then
        unrolledLoop ← unrollLoop(loop, tripCount);
        runConstProp(unrolledLoop);
        unrollCost ← evaluateCost(unrolledLoop);
        if $unrollCost > costThreshold$ then
            rerollLoop(unrolledLoop, tripCount);
        end
    end
    if not isConstant(tripCount) then
        peeledLoop ← peelLoop(loop, peelCount);
        runConstProp(peeledLoop);
        if isFullyUnrolled(peeledLoop) then
            unrollCost ← evaluateCost(peeledLoop);
            if $unrollCost > costThreshold$ then
                rerollLoop(peeledLoop, tripCount);
            end
        else
            rerollLoop(peeledLoop, tripCount);
        end
    else
    end
```

Figure 2a: The example parsing routine extracts the command-line options in a loop.

```c
void parse_input(int argc, char *argv[], struct Config *config){
  while ((int c = getopt (argc, argv, "pd")) != -1){
    switch (c){
      case 'p': config->enable_plugins = false; break;
      case 'd': config->enable_debug = false; break;
    }
  }
}
```

Once the loop body is fully unrolled, it allows for precisely determining the usefulness of the unroll. After unrolling, the `runConstProp` procedure runs a constant propagation pass on the unrolled loop. The purpose of invoking the constant propagation pass is to facilitate simplification of the unrolled loop expressions. Subsequently, `evaluateCost` leverages a cost model to compute the cost associated with unrolling the target loop. Since unrolling certain loops can potentially increase code size, it is necessary to evaluate the benefit of unrolling and skip unrolling if it does not assist constant folding of loop-oriented expressions. `evaluateCost` determines the fraction of instructions simplified as a result of applying constant propagation on the unrolled loop and accordingly computes a cost value. If no or few instructions are simplified/folded, the cost model returns a high cost value. If the cost model returns a high cost associated with the unroll, the unrolled loop is rerolled to regenerate the original loop. If the loop trip count is non-constant, the `peelLoop` procedure is invoked to peel the first few iterations of the loop. Subsequently, `runConstProp` invokes constant propagation to promote constant folding of expressions in the peeled iterations. Applying constant propagation on the peeled loop iterations potentially allows for statically evaluating the loop exit conditions, thereby assisting a full unroll. If the loop is not fully unrolled, it cannot promote further constant propagation, and hence, the loop is rerolled.

Figure 2b: The parsing loop is fully unrolled.

```c
void parse_input(int argc, char *argv[], struct Config *config){
    int option = 'p'; // getopt() replaced
    switch (option){
      case 'p': config->enable_plugins = false; break;
      case 'd': config->enable_debug = false; break;
    }
    option = 'd'; // getopt() replaced
    switch (option){
      case 'p': config->enable_plugins = false; break;
      case 'd': config->enable_debug = false; break;
    }
  }
}
```

Figure 2c: The unrolled `getopt` calls are folded to corresponding constant values.

```c
void parse_input(int argc, char *argv[], struct Config *config){
  config->enable_plugins = false;
  config->enable_debug = false;
}
```

Figure 2a shows an input parsing routine that reads the input arguments in a `getopt` loop and sets the program configuration variables accordingly. For purposes of illustration, we assume that the static program input includes two command-line flags, `p` and `d`. Since the loop exits when the `getopt` call indicates no more options are present, the trip count for the argument processing loop is not a compile-time constant. Accordingly, the unrolling transform peels the first few iterations of the loop, anticipating that the subsequent constant folding transform can allow for statically evaluating the loop exit condition, thereby effectively fully unrolling the target loop. The transformed program in Figure 2b shows the loop is fully unrolled particularly because the result of the `getopt` call is statically evaluated in terms of the static program arguments. As `getopt` and `getopt_long` are very commonly used in parsing code, we have included a custom transform to specialize these calls in terms of constant arguments. As the `getopt` calls are evaluated, the configuration settings can be correspondingly resolved to constant values. Figure 2c demonstrates the configuration settings are correspondingly resolved to constant values. The example demonstrates how fully unrolling the input parsing loop facilitates the constant folding of expressions in the loop body.

3.3 Constant Propagation

The goal of our tool is to effectively debloat unused application functionality with respect to a given user-defined usage context. In order to enable this, the user-provided constant configuration values must be aggressively propagated throughout the program callgraph, thereby aiding the compiler optimization passes in further simplifying program expressions and potentially pruning dead code. However, due to the conservative nature of constant propagation in production compilers such as LLVM [17], these simplifications are often not fully realizable. The conservative nature of these transforms is inspired by a need for maintaining reasonable compile-times in the general compilation workflow, thus avoiding potentially expensive analyses. Nonetheless, for our particular use case of debloating, a more sophisticated albeit relatively expensive constant propagation transform is justified. Our experimental results demonstrate the analyses overheads are very reasonable.

We develop an interprocedural constant propagation transformation that provides more precise propagation of configuration invariants. The transform works by maintaining memory state for each of the program objects and tracks the loads and updates from the target objects. Algorithm 3 gives the algorithm for the transform. 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 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:

Algorithm 3: Algorithm for Interprocedural Constant Propagation

```
Function processCallInst(callInst, context)
    if callee is externally defined then
        foreach argument in callInst do
            checkSideEffects(argument, context);
        end
    else
        runOnBasicBlock(callee → entryBlock, context);
    end
Function processBranchInst(branchInst)
    foreach successor block in branchInst do
        if all predecessors are visited then
            newContext = mergePredecessorContext();
            runOnBasicBlock(successor, newContext);
        end
    end

Function runOnBasicBlock(basicBlock, context)
    i = first instruction in basicBlock ;
    repeat
        if i is an allocation then
            objectContext ← createObjectContext(i);
            addToContext(objectContext, context);
        end
        if i is a store instruction then
            if constant value store then
                updateMemContext(operand, source, context);
            else
                markNonConstant(operand, context);
            end
        end
        if i is a load instruction then
            if operand is constant in context then
                replaceLoadWithConstant(i, context);
            end
        end
        if i is a call instruction then
            processCallInst(i);
        end
        if i is a branch instruction then
            processBranchInst(i);
        end
        i ← getSuccessorInst(i);
    until i is the last instruction;
    markVisited(basicBlock);
```

Handling Allocations: For each allocation site (Heap and Stack allocations), a context data structure is created to represent the memory state of the underlying memory object. The memory state for globally declared objects is created at analysis startup.

Handling Loads and Stores: For each `Store` instruction, the memory context of the target memory object is updated. For constant value stores, the memory state is updated with the corresponding constant value. For non-constant stores, the corresponding memory state is marked as non-constant. For each `Load` that corresponds to a constant value in the target memory context, the load is directly replaced with the constant value. Such constant folding of loads promotes further constant propagation.

Handling Function Calls: The `processCallInst` procedure details the policies for handling call instructions. For callee functions defined internally, the control is transferred to the callee. Moreover, the state of the memory context at the call-site is forwarded to the callee.

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. Therefore, for each distinct memory context, a new cloned specialized routine must be created. However, since creating multiple clones of a single function can increase code size, we only specialize functions that have a single identical memory context across all call-sites. This conservative approach prevents a code size blow up that could result with generating a specialized function per distinct memory context.

Handling Branches: As branch instructions are encountered, control is transferred to the successor blocks. The `processBranchInst` procedure details the policies for handling branch instructions. Control is transferred to a basic block once all its predecessor blocks have been visited. This 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. Specifically, the call to procedure `mergePredecessorContext` merges the memory contexts of all predecessor blocks. We include special handling of loops. If the basic block is the header (entry block) of a loop, the analysis does not wait for the analysis completion of the loop latches (blocks that branch back into the loop header) and starts traversing the loop body.

The implementation supports a much wider range of instruction types. However, due to space limitations, we have detailed only specific, interesting details of the algorithm. Figure 3a includes code that populates the configuration structures with the user-provided settings. The parsing code under `parse_input` is assumed to be already simplified by a combination of input specialization and loop unrolling. The routine `start_process` includes code that is conditional on the configuration settings. Since the configuration values `enable_plugins` and `enable_debug` are resolved as constants, the constant propagation transform can be leveraged for propagating the constant values to the corresponding branch conditions under `start_process`. This allows existing compiler optimizations to further simplify the code by statically evaluating the branch

Figure 3a: The example code invokes routines that are conditional on the configuration settings. The data structure `config` is populated with the configuration values.

```
void parse_input(int argc, char *argv[], struct Config *config){
  config->enable_plugins = false;
  config->enable_debug = false;
}
void start_process(struct Config *config){
  if(config->enable_plugins)
    load_plugins(); // conditional invocation
  if(config->enable_debug)
    print_debug(); // conditional invocation
  start_main_process(config);
}
void main(int argc, char *argv[]){
  struct Config* config = malloc(sizeof(struct Config));
  parse_input(argc, argv, config);
  start_process(config);
}
```

Figure 3b: The constant configuration values are propagated; hence, pruning the unreachable branch conditions, and in turn, eliminating unreachable routines `load_plugins` and `print_debug`.

```
void start_process(struct Config *config){
  // pruned unreachable calls
  start_main_process(config);
}
void main(int argc, char *argv[]){
  struct Config* config = malloc(sizeof(struct Config));
  parse_input(argc, argv, config);
  start_process(config);
}
```

conditions, thereby eliminating dead code. Figure 3b shows the corresponding transformed program after applying our constant propagation transform followed by the standard LLVM optimizations. The calls to routines `load_plugins` and `print_debug` are pruned as a result of the optimizations.

3.4 Soundness of Transformations

Trimmer provides propagation of configuration values through a combination of sequentially executed passes: input specialization, loop unrolling, 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.

Input Specialization: In the input specialization transform, we replace occurrences of the program input arguments (*argv* references) with corresponding constant values provided in the manifest file. Input 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 to reconfigure (e.g., HUP signal is commonly used in Unix programs to re-read the configuration file) [45]. 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.

External Function Calls: For external function calls with unknown semantics, we make conservative assumptions regarding the memory side effects. Notably, a) 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, and b) we assume arbitrary global memory side effects, marking all program globals 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 provide a) paths to library modules (as part of the input manifest file), or b) statically link the libraries with the application modules (eliminating external calls).

Indirect Calls: Since our current framework does not include support for devirtualizing indirect function calls, it makes conservative assumptions for memory side effects. Specifically, arguments passed to indirect function calls are conservatively assumed to be modified and marked non-constant. Moreover, since the targets of indirect calls cannot be precisely determined, we conservatively mark all program globals as non-constant. These conservative assumptions can limit the precision of constant propagation in scenarios where the configuration variables are globals. While an incomplete call-graph negatively impacts the precision of our analyses, it does not render it unsound. Since indirectly called functions have their addresses taken, the dead code elimination transforms in LLVM do not prune these functions.

4 EVALUATION

Our evaluation seeks to answer the following questions: (1) Does Trimmer effectively reduce code bloat for real-world programs? (2) Does Trimmer reduce the attack surface for code-reuse attacks? (3) What are the performance implications of specialization? (4) Are the analysis overheads reasonable?

4.1 Experimental Setup

To evaluate the effectiveness of our tool, we include four compilation pipelines.

- **Baseline:** The *Baseline* pipeline includes programs compiled with the standard compiler optimizations. Specifically, we compile applications at the -O2 level of optimization.

- **PE:** The second pipeline, referred to as *PE*, models the workflow of an LLVM-based partial evaluation tool *OCCAM* [40]. While *OCCAM* uses a similar approach of specializing applications for static arguments, it does not include the loop unrolling and constant propagation transforms for effectively propagating the static configuration. Our results show the lack of these transforms leads to missed opportunities for code debloating.
- **Trimmer:** This compilation pipeline represents the workflow of our debloating tool Trimmer illustrated in Figure 1. In addition to our proposed transforms, we apply the standard LLVM optimizations included in the -O2 optimization level.
- **Autotuned:** Program autotuning is an optimization technique that is extensively used to improve program performance [14, 23]. Autotuning leverages heuristic search space exploration to identify configurations that maximize a given objective. The configurations contain parameters and settings that impact the target objective. OpenTuner [4] is a popular tool that provides an extensible framework for autotuning by allowing users to specify the search space and the particular objective to optimize. For our experiments, we used OpenTuner to search for compiler pass sequences that optimize the binary size. Applying OpenTuner to a specialized program is useful since it allows for discovering optimal compiler pass sequences that better exploit the program constants. For this reference pipeline, we replace the -O2 optimization level with an autotuned compiler pass sequence extracted by OpenTuner (on a per-program basis).

4.2 Applications

For our experimental evaluation, we include 13 commonly used Linux applications. The program descriptions are detailed in Table 1. We selected a diverse set of programs including networking tools (knockd, httping, netperf, netstat, aircrack-ng, airtun-ng), popular Linux tools for data transfer, compression, profiling, and caching (curl, bzip2, gprof, memcached), commonly used Linux utilities (objdump, readelf), and an SMT solver (Yices). Overall, we select programs that provide configurable command-line flags that can be leveraged for program specialization. Currently, our tool does not include transforms for file I/O specialization and thus we do not include applications that read configuration files. Since some of these programs are also used in embedded systems, the code size reductions achieved have important implications for resource-constrained devices. In terms of reducing resource usage, while binary size reduction is not particularly valuable for general-purpose systems, a smaller code footprint reduces the exploitable vulnerabilities and simplifies security analysis. Section 4.4 discusses how a trimmed binary results in a reduced attack surface.

The corresponding static arguments used for specialization are also detailed in Table 1. The "__" symbols in the static arguments column indicate non-constant arguments that can be specified at runtime. For instance, the name of an input file may be provided as a runtime argument to the specialized binaries. Our selection of static program arguments is driven by two common scenarios. Specifically, we select arguments that i) represent the core functionality of the application while leaving out the auxiliary features, or ii) represent a common use case for an application that supports multiple use cases. The first scenario is useful with applications that provide certain core functionality, and the auxiliary features are rarely ever used and hence can be eliminated. The second scenario is useful with applications that have multiple usage scenarios, however, only a fraction of those usage scenarios are likely to be relevant in a particular deployment.

4.3 Reducing Code Bloat

In this section, we evaluate the code size reductions achieved by using Trimmer to specialize applications for a set of constant program arguments. Figure 4 shows the code size reductions for the included applications across the four reference compilation pipelines. Now we describe the use cases of specializing programs for static configurations. Due to space constraints, we detail a subset of specialization scenarios:

knockd is a popular port-knock tool used in Linux servers [39]. knockd listens to all traffic on an Ethernet (or PPP) interface, looking for special "knock" sequences of port-hits. A client makes these port-hits by sending a TCP (or UDP) packet to a port on the server. We specialized knockd for the more commonly used Ethernet interface (also the default), which resulted in slimming the binary by 23.8%. The code size reduction is achieved by removing the support for the PPP interface.

memcached is a popular key-value distributed memory object caching system. It is used extensively for caching results of database queries and API calls [29]. We specialized memcached for two essential arguments: max memory (-m) and the IP address to bind to (-l). Since the values corresponding to the max memory and IP address flags are specified as dynamic values ("__" symbols), these values can still be provided as arguments to the specialized binary. For this configuration, we observe a reduction of 14%. The reduction is achieved by debloating support for auxiliary features such as knobs that tune verbosity levels, page sizes, and data item sizes.

netperf is a benchmark tool used to determine the maximum latency and throughput between two endpoints [9, 11]. netperf provides support for a range of network tests for protocols including TCP and UDP. We specialize netperf for commonly used arguments including: IP of the remote end point (-H), name of network test (-t), intervals for displaying interim results (-D), and length of the test (-l). The values for these flags are passed as runtime values to the specialized binary. Therefore, the specialized binary still supports all included network tests while debloating auxiliary functionality including output formatting, debug output, and IPv6 connections among others. Specialization for this configuration provides a size reduction of 22.1%. Greater size reductions are achievable by specializing netperf for one particular network test. For instance, merely including support for the default "TCP_STREAM" test yields a reduction of 38.1%. For 11 network tests included with netperf, specializing for one network test provides a mean size reduction of 54.8%.

curl is a tool commonly used for transferring data. It is extensively used in cars, television sets, routers, printers, audio equipment, mobile phones, and media players among other software applications [50]. curl includes support for multiple protocols including HTTP, FTP, SMTP, IMAP, and LDAP among others. curl is a feature-rich program with more than 120 command-line options for various tasks and knobs including proxy connections, FTP operations, progress bars, rate limiting, and IPv6 addressing among others. While curl is used in a variety of different ways, it is

Table 1: Descriptions of the studied applications. The static arguments used for specialization are included. The "___" symbols indicate non-constant arguments that can be specified at runtime.

Application	Binary Size	Static Arguments	Description
knockd	47 KB	-i eth0	Listens to traffic on a specified network interface, looking for sequences of port hits
httping	44 KB	-G -s -X -b -B ___	Measures latency and throughput for a given webserver
netperf	162 KB	-H ___ -t ___ -D ___ -l ___	Benchmark for measuring various aspects of networking performance
aircrack-ng	113 KB	-b ___ -a wpa -s -w dictionary.lst ___	Used to assess WiFi network security
airtun-ng	88 KB	-a ___ -w ___	Creates a virtual tunnel interface to monitor encrypted network traffic
bzip2	31 KB	-force –keep –quiet –small ___	Popular compression tool used on major Linux distributions
objdump	1.9 MB	-D –syms -s -w ___	Displays information from object files. Options control the information displayed
readelf	524 KB	-h -l -S -s -r -d ___	Displays information about ELF files. The options control the information displayed
gprof	894 KB	-c -r -i -s ___	Profiles program execution and collects program statistics
memcached	100 KB	-m ___ -l ___	An in-memory key-value store for small chunks of arbitrary data
netstat	117 KB	-a -e -p -s	Prints network connections, routing tables, interface statistics, multicast memberships
curl	279 KB	–compress –http2.0 –ipv4 –ssl –url ___	Transfers data from or to a server, using one of the supported protocols
yices	1.6 MB	–logic=QF_AUFBV ___	SMT Solver with support for arithmetic, array, bitvector, and uninterpreted functions

unlikely a deployment scenario would require all the shipped functionality. In such cases, specialization can be leveraged to remove the undesired features. We specialized curl for the common usage scenario of reading data over an HTTPS connection given a target URL (curl-1 in Figure 4). This particular curl configuration is commonly used for reading RSS feeds and provides a 75.4% binary size reduction. Similarly, specializing curl as an SMTP client for sending emails (curl-2) reduces the binary size by 71.8%. These reductions are achieved by debloating protocols and features that were not required in the corresponding configurations. Since usage scenarios will vary across deployments, users can define static configurations that best represent their use cases.

Yices [21] is an SMT solver that includes support for four theories including: linear and non-linear arithmetic, theory of arrays, bitvector theory, and theory of uninterpreted functions. SMT solvers are used in a range of different application domains including hardware verification [15, 47], symbolic execution [13, 38], and bounded model checking (BMC) [8] among others. We specialized Yices for 3 different combinations of theory solvers to generate versions specialized for a particular problem domain. Since hardware verification usually requires SMT solvers with support for the bitvector theory [27], we specialized Yices for this configuration (yices-1 in Figure 4) yielding a 10.2% reduction in code size. Similarly, symbolic execution and test generation require support for the theory of bitvector and arrays [12, 22]. Specializing Yices for this configuration (yices-2) yielded a 7.7% reduction in binary size. Bounded model checking and k-induction for timed systems merely require support for linear real arithmetic [31]. Specializing Yices for the linear arithmetic solver (yices-3) resulted in a 24.1% size reduction.

Results Summary: As in the aforementioned use cases, the rest of the applications have been specialized with static configurations that represent realistic use cases. For the 13 programs studied, we observe a mean binary size reduction of 21.1% compared to the baseline. For all programs, partial evaluation (*PE*) using the OCCAM tool does not provide binary size reduction. Since OCCAM does not include the necessary simplifying transforms of improved loop unrolling and constant propagation, it misses opportunities for debloating code.

We leverage autotuning techniques to search for combinations of optimizations that reduce code size. In comparison to applying the standard -O2 optimization pipeline, autotuning the specialized application provides an additional 4.3% size reduction (25.4% overall) on average. These results show an optimized compiler pass sequence can better exploit the constants introduced as part of specialization.

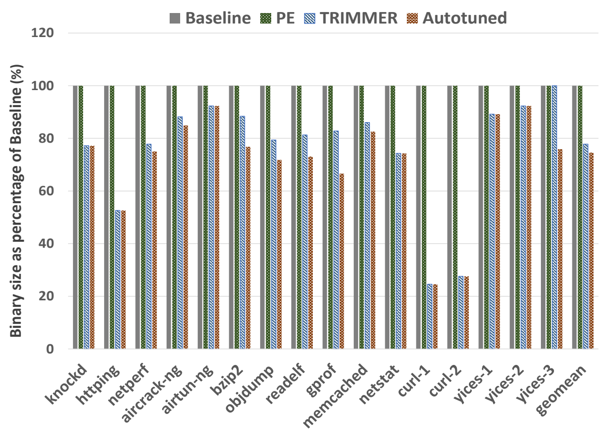

Figure 4: Comparing binary sizes across the four reference compilation pipelines. The reductions are shown relative to the *Baseline*.

4.4 Reducing the Attack Surface

We assess the implications of specialization on reducing code reuse attacks. A code reuse attack is more powerful compared to code injection attacks since it leverages existing code snippets (called gadgets) in the executable as the attack payload [46]. These attacks require subverting the programs' control flow by exploiting vulnerabilities such as buffer overflows [3], integer overflows [42], or format string vulnerabilities [37]. Gadgets are small code sequences that usually end with a `RET` or `JMP` instruction. This allows attackers to chain pointers to the gadgets on the stack, thereby constructing a malicious payload. However, to conduct a code reuse attack, an attacker must have knowledge of the gadget locations in the executable [32]. Prior work has focused on diversification techniques that limit the attacker to a small subset of these predictable targets, thus rendering the attacks infeasible. Some of these techniques include NOP insertion and instruction location randomization (ILR) [25, 26].

Our results with TRIMMER show that specialization can reduce the number of exploitable gadgets as well as relocate the gadget locations, thus diversifying the binary. While we do not claim specialization serves as a standalone defense mechanism, we show it can significantly reduce the attack surface. Table 2 shows the results of our gadget analysis. For our experiments, we used ROPgadget [44], a popular tool for finding gadgets in program binaries. The *Gadgets Baseline* column includes the total number of gadgets in the original binary compiled with the baseline. The *Gadgets TRIMMER* column includes the number of gadgets in the binary after applying specialization using TRIMMER. For all applications, we observe noticeable reductions in the total gadget count. Across all programs, we observe a mean reduction of 20.1% and a maximum reduction of 87% (for curl). We also measure the system call gadgets across the original and specialized binaries. To carry out useful tasks, attackers often need to exploit system calls [10]. Hence most ROP attacks set up the stack to execute short sequences of gadgets before a system call is issued [19]. The gadgets executed prior to the system call gadget allow for setting up the stack with the intended arguments, thus allowing an attacker to execute an operation of interest. Our results indicate for 7 out of the 8 programs that contain system call gadgets, the gadgets are reduced. For 4 of these programs, the system call gadgets are completely removed, therefore creating binaries that are not susceptible to ROP attacks that leverage system calls. Notably, specializing *curl* for both configurations (described in the previous section) eliminated all 11 system call gadgets in the baseline binary. For 3 programs, we observe a slight increase (of 1) in system call gadgets. Although specialization may potentially introduce new gadgets, we observe that debloating unused features mostly reduces gadgets.

Specializing applications for specific configurations allows for diversifying the program, making it harder for attackers to gain insights about the gadget locations. To measure diversification, we count the number of functionally equivalent gadgets at the same byte location in the baseline and specialized binaries. We call these *surviving gadgets*, consistent with prior work in code diversification [26]. Notably, across all programs, the surviving gadgets are reduced to 0, showing that instruction sequences are sufficiently modified by our transforms. The benefits of diversification are enhanced if an application can be specialized for a range of different configurations and each specialized binary is sufficiently dissimilar from the other. To evaluate this, we specialized the Yices SMT solver for different combinations of theory solvers. For all 12 possible configurations of Yices, we generated a specialized binary for each configuration and did a pair-wise comparison of common gadgets (functionally equivalent gadgets at the same byte location) across the binaries. The highest gadget overlap across any two binaries is only 4.2%. Moreover, none of the 12 specialized binaries has any common gadgets with the baseline binary. Similarly, comparing the specialized binaries corresponding to 5 different configurations of curl, the highest gadget overlap is only 3% and none of the specialized binaries has any common gadgets with the baseline binary.

Compared to existing diversification and enforcement-based mechanisms, specialization provides a major advantage by incurring no performance overhead and reducing space overhead. Techniques such as CFI, ILR, NOP insertion all have negative implications for code size and performance [1, 25, 26, 53]. Overall, TRIMMER can provide a useful diversification mechanism for applications that can be specialized based on their deployment settings. Diversification via specialization can be used in conjunction with enforcement-based mechanisms such as CFI to provide stronger security guarantees.

Table 2: "Gadgets Baseline" refers to the total gadgets in the original binary, while "Gadgets TRIMMER" refers to the gadgets remaining in the specialized binary (applying TRIMMER). "Syscall Gadgets" is the number of system call gadgets in the original binary, while "Syscall Gadgets TRIMMER" refers to gadgets remaining in the specialized binary. Reduction (%) shows the percentage reduction in the total gadgets after specialization.

Application	Gadgets Baseline	Syscall Gadgets	Gadgets TRIMMER	Syscall Gadgets TRIMMER	Reduction (%)
knockd	246	0	161	0	34.6 %
httping	580	0	218	0	62.9%
netperf	963	1	793	0	17.7%
aircrack-ng	1091	0	955	1	12.5%
airtun-ng	786	1	733	0	6.7%
bzip2	246	0	208	0	15.5%
objdump	11228	1	9419	2	16.1%
readelf	3875	1	2902	0	25.1%
gprof	9555	6	8433	1	11.7%
memcached	883	0	641	1	27.4%
netstat	440	4	355	0	19.3%
curl-1	2786	11	372	0	86.7%
curl-2	2786	11	382	0	86.2%
yices-1	12245	8	11348	3	7.3%
yices-2	12245	8	11651	3	4.9%
yices-3	12245	8	9660	3	21.1%

4.5 Performance Impact

To understand the performance impact of application specialization, we measured the execution time of each program over ten runs, comparing the original and specialized binaries. For gprof, curl, and netperf, we observe noticeable performance improvements of 4.8%, 4.6%, and 13%, respectively. For the remaining programs, we observe less than 1% improvement. Measuring the dynamic instructions shows specialization improves performance when the instructions executed on the hot path of execution are reduced. In other instances, although debloating reduces the static instructions, it does not significantly reduce the instructions executed on the hot path. Prior work has shown autotuning compiler transforms for parallelizing compilers can yield significant performance improvements [5, 51]. However, since our autotuning search is tuned for minimizing code size, we do not observe noticeable performance differences compared with the -O2 pipeline.

4.6 Analysis Time

We evaluated the time usage of our tool on a Linux workstation with an Intel Core i5-3380M CPU running at 2.9 GHz. We compiled our LLVM transforms with Clang at the -O2 level of optimization. Figure 5 shows the percentage breakdown of the individual transform overheads. *Loop Unroll* denotes the overhead for the loop unrolling transform, *Const Prop* shows the overhead for the constant propagation transform, and *O2* denotes the overhead of running the

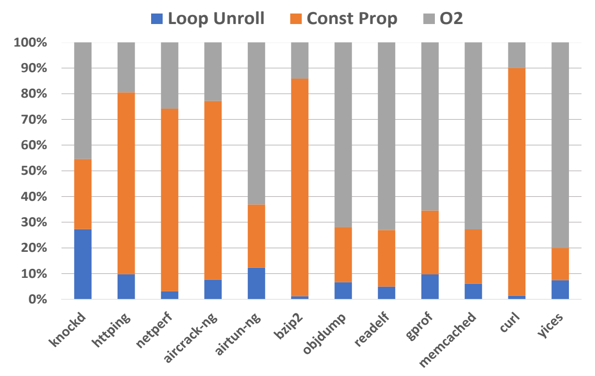

Figure 5: Comparing the analysis overheads of loop unrolling, constant propagation, and the standard LLVM optimizations (-O2).

LLVM -O2 optimization pipeline, which includes multiple standard compiler optimization passes. We do not show the execution times for the input specialization transform since that incurs negligible overhead. The evaluation indicates loop unrolling does not incur significant overhead, while the -O2 optimization pipeline incurs the largest fraction of the analysis time. On average, loop unrolling constitutes 6.4%, constant propagation constitutes 35.5%, and -O2 constitutes 58.1% of the total execution time. The absolute time usage of the tool is also reasonable. On average, the analysis time across all programs is 7.2 seconds, and the maximum analysis time (for Yices) is 18.9 seconds. The time usages are quite reasonable, particularly considering the specialization overhead is a one-time cost. These overheads indicate it may be feasible to incorporate more sophisticated loop unrolling and constant propagation transforms as part of the LLVM toolchain.

5 RELATED WORK

Program specialization has been the topic of both earlier and recent research efforts. Our tool builds on previously published work in the area of specialization. We see our work fitting in the broad scope of these efforts, with a specific motivation for code debloating.

Malecha et al. [40] presented a partial evaluation toolchain for specializing whole programs. While their tool includes transforms for specializing programs with respect to the predefined static user input, it entirely relies on the standard LLVM optimizations for providing the necessary code simplification. In our work, we observe these techniques are inadequate for providing effective specialization. Instead of relying on heroic compiler analysis, we develop transforms that improve constant propagation.

Smowton et al. [48] proposed a framework that propagates constant data from the file system into the program. These techniques are particularly relevant for programs that read static configuration files. While these techniques present interesting solutions, we believe they need to be extended to work effectively with real-world programs. As part of future work, we hope to investigate how to integrate file-based specialization as part of our framework.

Specialization has also been envisioned in the broader context of optimizing the full software stack [6, 41]. Bhatia et al. [6] applied partial evaluation techniques for specializing the network stack in operating system kernels. The approach applied is semi-automatic as it requires the user to provide declarative annotations for specifying program invariants. We believe this process can be fully automated and thereby made more practical, by extending our tool with the necessary techniques for handling OS-specific constructs.

Debloating application containers such as those provided by Docker has been the topic of recent work [43]. Rastogi et al. employ an approach based on dynamic analysis for automatically partitioning containers to provide better privilege separation across different applications. In particular, dynamic analysis is leveraged to identify the resource dependencies of applications thereby guiding the partitioning process. However, their tool only partitions containers at the granularity of individual executables and does not partition executables. TRIMMER uses static analysis techniques to debloat functionality at finer granularity.

6 LIMITATIONS AND FUTURE WORK

TRIMMER provides an important step in making specialization practical for real-world programs. In this section, we point out some limitations of our tool and discuss directions for future research.

Operating system stacks often contain low-level assembly code to implement the architecture-specific operations. To specialize the full software stack, compiler transforms must either have the capability to analyze assembly code or rely on the translation of assembly code to the compiler IR [20]. As part of future work, we intend to extend TRIMMER with analyses that can reason about assembly code. Moreover, systems code and applications often include indirect function calls that can limit the capabilities of static analysis [30]. In the presence of indirect function calls, our constant propagation transform makes conservative assumptions about memory side effects. In future research, we hope to include analyses for devirtualizing indirect function calls, thereby facilitating more precise constant propagation. Currently, our tool specializes applications for static program arguments. To support applications that read configuration files, we intend to build transformations for specializing file I/O operations.

7 CONCLUSION

In this paper, we introduced TRIMMER, an application specialization tool that debloats unused functionality by specializing programs for user-defined configurations. Our results demonstrate our tool effectively optimizes binary sizes for real-world programs. For applications spanning various domains, we observed a mean binary size reduction of 21% and a maximum reduction of 75%. We also showed specialization can serve as a mechanism for introducing software diversity and can reduce the number of exploitable gadgets. For the evaluated programs, we observe a 20% mean reduction in the total gadgets and a maximum reduction of 87%. We believe our tool is an important step towards the goal of specializing the full software stack.

ACKNOWLEDGEMENTS

This material is based upon work supported by the US National Science Foundation (NSF) under Grant ACI-1440800 and the Office of Naval Research (ONR) under Contract N68335-17-C-0558, Grant N00014-4-1-0525, and Grant N00014-17-1-2996. 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] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (2005), ACM, pp. 340–353.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers, Principles, Techniques.* Addison Wesley Boston, 1986.

[3] Aleph, O. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14* (1996).

[4] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)* (2014), IEEE, pp. 303–315.

[5] Basu, P., Venkat, A., Hall, M., Williams, S., Van Straalen, B., and Oliker, L. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *20th Annual International Conference on High Performance Computing* (2013).

[6] Bhatia, S., Consel, C., Le Meur, A.-F., and Pu, C. Automatic specialization of protocol stacks in OS kernels. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks* (2004).

[7] Bhattacharya, S., Rajamani, K., Gopinath, K., and Gupta, M. The interplay of software bloat, hardware energy proportionality and system bottlenecks. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems* (2011), ACM, p. 1.

[8] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., Zhu, Y., et al. Bounded model checking. *Advances in Computers 58* (2003), 117–148.

[9] Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K. Network-oriented full-system simulation using m5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)* (2003), pp. 36–43.

[10] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 30–40.

[11] Blum, R. *Network Performance Open Source Toolkit: using Netperf, tcptrace, NISTnet, and SSFNet.* John Wiley & Sons, 2003.

[12] Brummayer, R., and Biere, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2009), Springer, pp. 174–177.

[13] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)* (2008), vol. 8, pp. 209–224.

[14] Christen, M., Schenk, O., and Burkhart, H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (2011), IEEE, pp. 676–687.

[15] Clarke, E., Talupur, M., Veith, H., and Wang, D. SAT based predicate abstraction for hardware verification. In *International Conference on Theory and Applications of Satisfiability Testing* (2003), Springer, pp. 78–92.

[16] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.-N., Lawall, J., and Noyé, J. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys (CSUR) 30*, 3es (1998), 19.

[17] Cooper, K., and Torczon, L. *Engineering a Compiler.* Elsevier, 2011.

[18] Danvy, O. Type-directed partial evaluation. In *Partial Evaluation.* Springer, 1999, pp. 367–411.

[19] Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium* (2014), pp. 401–416.

[20] Dinaburg, A., and Ruef, A. Mcsema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada* (2014).

[21] Dutertre, B. Yices 2.2. In *International Conference on Computer Aided Verification (CAV)* (2014), Springer, pp. 737–744.

[22] Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV)* (2007), Springer, pp. 519–531.

[23] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., and Cavazos, J. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)* (2012), IEEE, pp. 1–10.

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

[25] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. ILR: Where'd my gadgets go? In *IEEE Symposium on Security and Privacy (SP)* (2012), IEEE, pp. 571–585.

[26] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., and Franz, M. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2013), IEEE Computer Society, pp. 1–11.

[27] Jha, S., Limaye, R., and Seshia, S. A. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *International Conference on Computer Aided Verification (CAV)* (2009), Springer, pp. 668–674.

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

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

[30] Kim, H., Joao, J. A., Mutlu, O., Lee, C. J., Patt, Y. N., and Cohn, R. VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. *ACM SIGARCH Computer Architecture News 35*, 2 (2007), 424–435.

[31] Kindermann, R., Junttila, T., and Niemelä, I. SMT-based induction methods for timed systems. In *International Conference on Formal Modeling and Analysis of Timed Systems* (2012), Springer, pp. 171–187.

[32] Larsen, P., Brunthaler, S., Davi, L., Sadeghi, A.-R., and Franz, M. Automated software diversity. *Synthesis Lectures on Information Security, Privacy, & Trust 10*, 2 (2015), 1–88.

[33] Larus, J. Spending Moore's dividend. *Communications of the ACM 52*, 5 (2009), 62–69.

[34] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (2004), IEEE Computer Society, p. 75.

[35] Lee, C.-T., Lin, J.-M., Hong, Z.-W., and Lee, W.-T. An application-oriented Linux kernel customization for embedded systems. *J. Inf. Sci. Eng. 20*, 6 (2004), 1093–1107.

[36] Lekatsas, H., and Wolf, W. Code compression for embedded systems. In *Proceedings of the 35th Annual Design Automation Conference (DAC)* (1998), ACM, pp. 516–521.

[37] Lhee, K.-S., and Chapin, S. J. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience 33*, 5 (2003), 423–460.

[38] Ma, K.-K., Phang, K. Y., Foster, J. S., and Hicks, M. Directed symbolic execution. In *International Static Analysis Symposium (SAS)* (2011), Springer, pp. 95–111.

[39] Madia, A., Nikoletseas, S., Stamatiou, Y., Tsolovos, D., and Vlachos, V. Crowd sourcing based privacy threat analysis and alerting. *Cryptography, Cyber Security and Information Warfare (3rd CryCybIW)* (2016).

[40] Malecha, G., Gehani, A., and Shankar, N. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)* (2015), ACM, pp. 1504–1511.

[41] McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., and Marlet, R. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS) 19*, 2 (2001), 217–251.

[42] Molnar, D., Li, X. C., and Wagner, D. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *USENIX Security Symposium* (2009), vol. 9, pp. 67–82.

[43] Rastogi, V., Davidson, D., De Carli, L., Jha, S., and McDaniel, P. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)* (2017), ACM, pp. 476–486.

[44] Salwan, J. Ropgadget tool. http://shell-storm.org/project/ROPgadget/, 2012.

[45] Schilit, B. N., Theimer, M. M., and Welch, B. B. Customizing mobile applications. In *Proceedings USENIX Symposium on Mobile & Location-indendent Computing* (1993), vol. 9.

[46] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)* (2007), ACM, pp. 552–561.

[47] Sheeran, M., Singh, S., and Stålmarck, G. Checking safety properties using induction and a sat-solver. In *International Conference on Formal Methods in Computer-aided Design (FMCAD)* (2000), Springer, pp. 127–144.

[48] Smowton, C., and Hand, S. Make world. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)* (2011), USENIX Association, pp. 26–26.

[49] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 574–588.

[50] Stenberg, D. Everything curl. https://legacy.gitbook.com/book/bagder/everything-curl/, 2017.

[51] Tiwari, A., Chen, C., Chame, J., Hall, M., and Hollingsworth, J. K. A scalable auto-tuning framework for compiler optimization. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2009), IEEE, pp. 1–12.

[52] Xu, G., Mitchell, N., Arnold, M., Rountev, A., and Sevitsky, G. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (2010), ACM, pp. 421–426.

[53] Zhang, M., and Sekar, R. Control flow integrity for COTS binaries. In *USENIX Security Symposium* (2013), pp. 337–352.