Reinforcement Learning Guided Software Debloating*

Nham Le Van **Ashish Gehani** **Arie Gurfinkel** **Susmit Jha** **Jorge A. Navas**

Abstract

Modern software engineering practices encourage software bloat. Libraries provide more APIs than any one client needs; programs have many command line and configuration options, with most never used by the user and remaining set to their default values. The result is software systems that are bloated, slow, and, most importantly, vulnerable to attacks. *Software debloating* is an emerging technique to address this problem by automatically trimming (or specializing) a program to a user-defined execution environment (e.g., command line options, configuration files, common usage scenarios, etc.). Since deciding on what to specialize is computationally expensive, current state-of-the-art software debloaters rely on hand-crafted heuristics. This is problematic because these heuristics are not always effective, and, more importantly, difficult to adjust to different debloating metrics. In this paper, we propose a reinforcement learning based approach to automatically learn a good debloating policy. We have implemented the approach as a tool called DEEPOCCAM, and evaluated it on debloating several C programs. Our results show that the technique outperforms the base-line policy of a state-of-the-art debloater OCCAM, while being easily adaptable to a variety of debloating metrics.

1 Introduction

The rapid increase of software productivity in the last decades was fueled by the extensive use of abstraction and reuse of software components. While this enabled building larger and more complex software systems, these gains came at the expense of less efficient and less secure software systems, and contribute to a troublesome trade-off between productivity and performance/security. When an application built using general-purpose components is deployed, the majority of the general-purpose functionality is never used. This creates two problems: first, as the use of abstraction layers makes it more difficult to optimize the software, this has a detrimental impact on performance; second, it increases the attack surface for security vulnerabilities.

An emerging solution to this problem is a set of tools, called *Software Debloaters*, that automatically customize a program to a user-specified environment. One successful approach for software debloating is based on partial evaluation (PE) [12] in which a *partial evaluator* takes a program and some of its input values and produces a *residual* (or *specialized*) program in which those inputs are replaced with their values. While PE has been extensively applied to functional and logic programs, it was less successful on imperative C/C++ programs (with a noteable exception of C-MIX [3] and TEMPO [9]). With the advent of LLVM [15], several new partial evaluators of LLVM bitcode have arisen during the last few years (e.g., LLPE [21], OCCAM [16], and TRIMMER [20]). These tools leverage LLVM optimizations such as constant propagation, function inlining, and others to either reduce the size of the residual program (e.g., OCCAM and TRIMMER) or improve performance (e.g., LLPE).

The key step of a PE-based debloater is to estimate whether specializing (or inlining) a function will result in a smaller (or more secure) residual program. Specialization naturally increases the size of

*This material is based upon work supported by the National Science Foundation under Grants ACI-1440800 and CNS-1740079. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Callee	Caller	Module	Call-Site
Basic blocks	Basic blocks	Functions	Arguments
Instructions	Instructions	Instructions	Known arguments
Store instructions	Store instructions	Basic blocks	
Call instructions	Call instructions	Direct calls	
Branch instructions	Branch instructions		
Loops	Loops		
Uses	Uses		
Influenced branches	Untouched call-sites		
Influenced instructions			

Table 1: RL features. *Influenced instructions (branches)* are the those whose operands have statically known arguments. *Uses* indicates how many times the *caller* and the *callee* are invoked.

the code since it adds extra functions. However, since some inputs in the new functions have been replaced by constant values, optimizations such as constant propagation, might become enabled and result in new optimization opportunities that reduce the code size. Therefore, the decision whether or not a function should be specialized for a particular call-site is non-trivial. Current PE-based software debloaters use naive heuristics. For example, OCCAM implements two heuristics of "never specialize" or "always specialize", respectively; TRIMMER specializes a function only if it is called once in the whole program.

In this paper, we present a new approach based on reinforcement learning (RL) to automatically infer effective heuristics for specializing functions for PE-based software debloating. RL is a good fit for the problem for two reasons. First, code specialization resembles a Markov Decision Process: each decision moves the code from one state to another, and specialization depends only on the current state rather than the history of the transformations. Second, the quality of debloating can only be measured after all specialization actions and corresponding compiler optimizations have been performed.

The main challenge in applying RL is in deciding on a good state representation. While it is tempting to use the source code (or LLVM intermediate representation (IR)) as a state, this is not computationally tractable. The IR is typically hundreds of MBs in size. Instead, an adequate set of features that captures meaningful information about the code while avoiding state aliasing is required. In particular, these features must capture the *calling context* in order to distinguish between call-sites.

The main contributions of this paper are threefold: (1) calling context features that enable RL to find a useful heuristic for PE-based debloating software; (2) implementation of our method in a tool called DEEPOCCAM; and (3) evaluation on the reduction of the program size and number of possible code-reuse attacks, by comparing our handcrafted features with features learnt automatically via embedding LLVM IR into a vector space using INST2VEC [5]. Our initial evaluation suggests that RL is a viable method for developing an effective specialization policy for debloating, and learning with handcrafted features is easier than with INST2VEC.

2 Methods

In this section, we formulate the debloating problem as a reinforcement learning problem, describe the learning procedure, and its hyperparameters.

Action, state, and reward. To determine whether a call-site should be specialized or not, we train a policy using reinforcement learning. This requires defining *action*, *state*, and *reward*.

An action is whether to run a code transformation, called *specialization*, or not. Consider a call to function $calleeF$ with arguments A at a call-site c_i. Let $\mathsf{formals}$ be a map from call-site arguments to their corresponding formal parameters. Let B, $B \subseteq A$, be the arguments whose values are are statically known. Then, specialization of the call-site does:

1. Create a new function $spec_calleeF(\mathsf{formals}(A \setminus B))$, such that the body of $spec_calleeF$ is identical to the body of the original $calleeF$ except that $\mathsf{formals}(B)$ are replaced with their values.
2. Perform constant propagation to push forward the information to the rest of callee's body, potentially specializing more call-sites in the callee.
3. Replace $calleeF(A)$ with $spec_calleeF(A \setminus B)$ at the call-site c_i.

Note that after specialization, a software debloater can trigger other optimization passes such as inlining and dead-code elimination.

A state is a vector capturing all the relevant information about the code. We experiment with two state representations: (a) a vector of hand crafted features (**HF**), and (b) an embedding of LLVM IR via INST2VEC (**IV**).

In **HF**, the state is a concatenation of four feature vectors, summarized in Table 1. They capture relevant information about the *callee*, *caller*, *compilation unit (module)*, and *call-site*. Most features are self-explanatory, and hence, we focus only on those which are novel or more relevant for avoiding state aliasing. The *state aliasing* problem occurs when two different states have the same representation in the RL model. In our context, our features must distinguish between two call-sites in the same basic block when invoking the same callee with the same arguments if one is specialized and the other not. In [14], the features *InLoop* (whether a call-site is in a loop), *InlineDepth* (the current inlining depth), and *currentGraphSize* (how many instructions are there in the caller, including one added by inlining), are used to decide inlining. However, in our case, these features are insufficient to prevent state aliasing. Thus, we also add *Untouched call-sites* that counts the number of call-sites yet to be processed. Since the call-sites are processed in a fixed order, *Untouched call-sites* is sufficient to distinguish any two call-sites in a function.

In **IV**, we use INST2VEC embedding from [5]. We extract the LLVM IR of the *caller*, the *callee*, and the *calling context* (a window of n instructions around the call-site) as a lists of instructions. Each instruction is embedded into a vector space using INST2VEC. Additionally, we encode the arguments at the call-site as a bitvector, in which each statically known argument is encoded by 1, and an unknown argument by 0. This bitvector is then embedded using a different embedding matrix. Finally, the **IV** state is a tuple of the above four 2-D matrices. Note that in this case, the calling context of each call is explicitly represented by the IR.

For rewards, we focus on two different metrics. First, we measure the number of instructions in the final binary produced by the software debloater after specialization took place. Second, we measure the reduction of the *attack surface*, focusing on code reuse attacks.

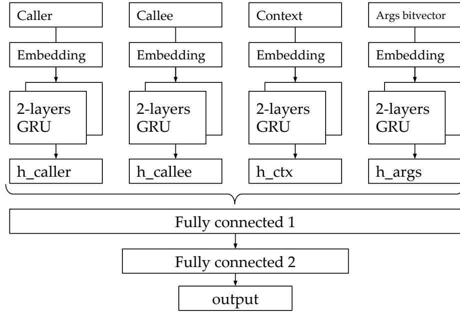
Code reuse attacks are exploits in which an attacker makes use of the available instructions in the binary to chain together short sequences called *gadgets*, and use those gadgets to compromise the control flow of the program, causing a malicious effect. Those sequences are often categorized based on their last instruction, into ROP (return-oriented programming), JOP (jump-oriented programming), and COP (call-oriented programming) gadgets, respectively. Since the relationship between the number of gadgets and exploitability is an open question [6], we focus on reducing the number of gadgets without making any further claim about the security of the debloated code.

The rewards are the negations of the number of instructions, ROP, JOP, and COP gadgets. The negation is necessary because we are interested in minimizing our metrics. We only have one measurement at the end of an episode (when the final binary is produced). Immediate rewards after each action are set to zero.

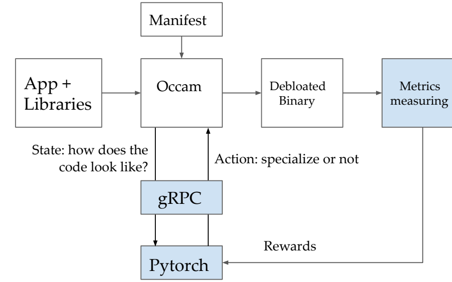
Learning procedure, policy network, and hyper-parameters. Each state representation requires a different neural net for the policy network. For **HF**, we use a 3-layer fully connected network. Before training, we run the pipeline with a random policy 100 times to calculate the mean and standard deviation of each feature, and use this metadata to normalize all features into mean of 0 deviation of 1.

For **IV**, we run the *caller*, the *callee*, the *calling context* and the *arguments bitvector* through four separate 2-layer GRU [8] blocks, concatenate the last hidden outputs of these 4 blocks, and then feed it to a 3-layer fully connected network. The architecture is depicted in Fig. 1a.

Both networks use ReLU [17] as the activation function. We use REINFORCE [22] with normalized rewards to update the policy for both models. At each REINFORCE iteration, we roll out k runs of the current policy, batch them together, and use the Adam optimizer [13] to update the network. For all metrics, we use the same hyper-parameters: INST2VEC calling context $n = 10$, number of runs in each policy rollout $k = 75$, Adam's learning rate = 0.001, and train up to 340 iterations.



(a) Policy network architecture using INST2VEC.



(b) DEEPOCCAM architecture. Boxes in blue correspond to this paper.

3 Implementation and Evaluation

We have implemented our prototype, called DEEPOCCAM, using OCCAM [16]. The architecture of DEEPOCCAM is depicted in Fig. 1b. DEEPOCCAM takes as inputs a set of LLVM modules (main application and libraries) and a manifest (i.e., a user-defined execution environment) and produces a specialized binary. We used Gadget Set Analyzer (GSA) [6] to evaluate the specialized binary and PYTORCH [18] for training the deep learning models.

At each call-site, OCCAM calculates the features described in Table 1 from the LLVM module, sending them to the PYTORCH server, and transforming the code based on the returned decision. During inference, the PYTORCH server receives the message and runs the learnt policy to decide whether to specialize that call-site. During training, the PYTORCH server also keeps track of all the decisions it has made and uses that information to update its policy. We run multiple copies of OCCAM on multiple copies of the PYTORCH server to scale up the learning process, which is justified by the Markov property of the state.

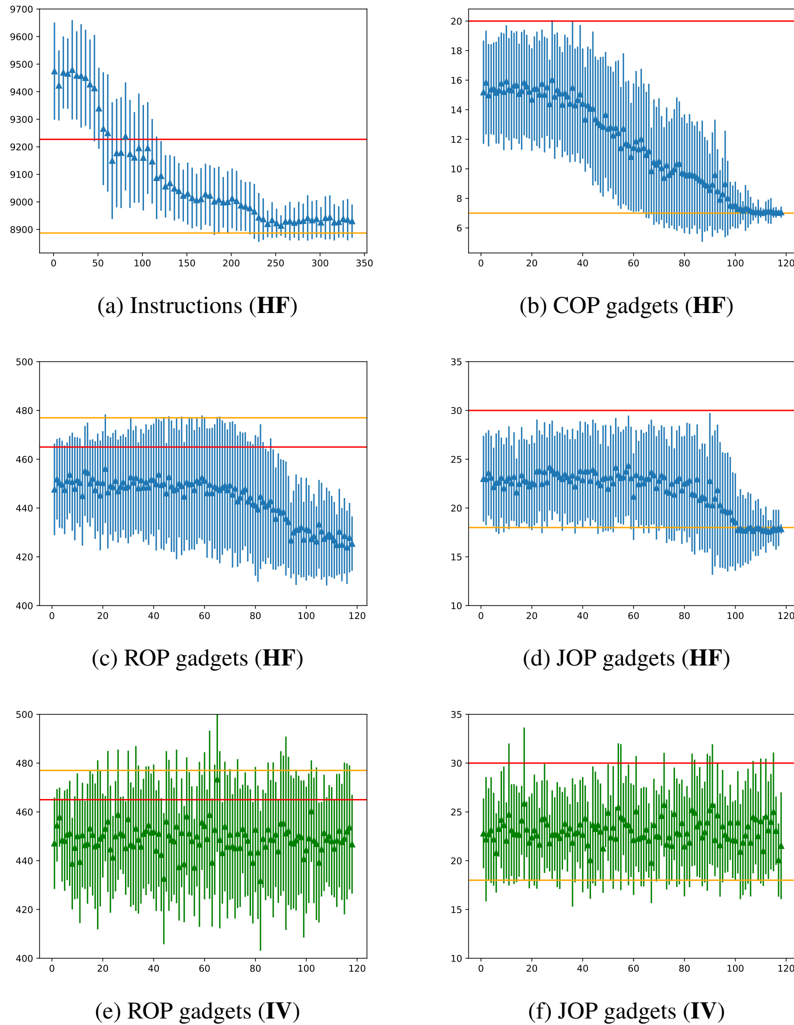
We compare DEEPOCCAM with OCCAM running in two modes. First, OCCAMAGG that runs OCCAM with a *nonrecursive-aggressive* policy. This always specializes a call-site if the callee is not a recursive function. Second, OCCAMNONE that runs OCCAM without specializing any call-site. Fig. 2 shows the comparision between DEEPOCCAM using **HF** , DEEPOCCAM using **IV**, OCCAMAGG, and OCCAMNONE for debloating GNU TREE. On average, DEEPOCCAM outperforms both OCCAMNONE and OCCAMAGG in reducing the number of ROP and JOP gadgets, matches OCCAMNONE on COP gadgets, and underperforms in reducing the number of instructions. Interestingly, while DEEPOCCAM matches OCCAMNONE on COP, it finds a different policy that specialize some call-sites. For optimizing the number of instructions, we run 200 more training iterations but are not able to outperform OCCAMNONE. We include the results for the number of instructions only to demonstrate the flexibility of our approach.

In all of our experiments, the learning procedure does not converge when we use INST2VEC pre-trained embedding. Upon closer inspection, we observe that the software in our test suite, once compiled to LLVM IR, contained many instructions regarded as low-frequency by INST2VEC. Consequently, they are all mapped into the same `UNK!` token in the cutoff dictionary. Hence, the calling context window that we use suffers from the state aliasing problem: different calling contexts are mapped to the same vector of tokens.

This is surprising since INST2VEC claims to train the embedding on a wide range of software written in C, C++, FORTRAN, and OpenCL, specifically to avoid overfitting to a small family of code bases.

4 Related work and Conclusions

Recent advances in deep learning and RL have opened new frontiers to the design of compiler heuristics [4]. Most current approaches define actions at the compilation unit level: whether to run a particular optimization pass, or how to schedule optimization passes. For example, Cummins et al. [10] model code as a natural language problem to predict whether to run the code on CPU or GPU, as well as the optimal GPU thread coarsening factors. Kulkarni et al. [14] use NEAT — a genetic algorithm — to learn an inlining heuristic for MaxineVM as a neural network. For interpretability,



(a) Instructions (**HF**) (b) COP gadgets (**HF**)

(c) ROP gadgets (**HF**) (d) JOP gadgets (**HF**)

(e) ROP gadgets (**IV**) (f) JOP gadgets (**IV**)

Figure 2: Results for optimizing different metrics using **HF** and **IV**. x-axis is the number of RL iterations. y-axis is the results. Dots and bars are mean and std. dev. of k runs in each iteration. **HF** results are in blue while **IV** results are in green. Orange and red lines are OCCAMNONE and OCCAMAGG results, respectively.

they then approximate the neural network by a decision tree. We, on the other hand, use actions at much lower granularity, focusing on specialization at each call-site.

Program code is a rich structure that can be presented in a variety of ways, e.g., as raw text, control-flow and call- graphs, etc. Recent application of deep learning for code experiment with models based on techniques from Natural Language Processing and Graph Neural Network (e.g., [2, 5, 1, 7]). Among them, INST2VEC [5] is the only one that works on LLVM IR.

The closest related work is Chisel [11] – a software debloater based on RL. Inspired by C-Reduce [19], Chisel takes a program P and a property of interest φ (i.e., P must compile and pass tests defining φ) and produces the smallest program P' that satisfies φ. RL accelerates the search for the reduced program providing a scalability boost. Compared to DEEPOCCAM, Chisel has very different state and action space, and, is generally only as sound as the test cases defining φ.

In this paper, we present DEEPOCCAM— an end-to-end tool to learn specialization heuristics for software debloating. Our preliminary results suggest that it is feasible to use RL to learn an effective specialization heuristic to optimize a variety of related metrics. They also suggest that out of the box, pretrained embedding such as INST2VEC might not be applicable for the task. We hope that

DEEPOCCAM contributions in feature engineering and architecture might be applicable to other compiler optimization tasks such as inlining.

References

[1] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *CoRR*, abs/1808.01400, 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. pages 40:1–40:29, 2019.

[3] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[4] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, 2019.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336, 2018.

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

[7] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *CoRR*, abs/1904.03061, 2019.

[8] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[9] Charles Consel, Luke Hornof, Renaud Marlet, Gilles Muller, Scott Thibault, E-N Volanschi, Julia Lawall, and Jacques Noyé. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.

[10] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, Sep. 2017.

[11] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *CCS*, pages 380–394, 2018.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[14] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Doug Simon. Automatic construction of inlining heuristics using machine learning. In *CGO*, pages 9:1–9:12, 2013.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–, 2004.

[16] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. Automated software winnowing. In *SAC*, pages 1504–1511, 2015.

[17] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[19] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[20] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *ASE*, pages 329–339, 2018.

[21] Christopher S.F. Smowton. I/O Optimisation and elimination via partial evaluation. Technical Report UCAM-CL-TR-865, University of Cambridge, Computer Laboratory, December 2014.

[22] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.