Program Debloating via Stochastic Optimization

Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso
Georgia Institute of Technology
{qxin6,wardballoon,qrzhang}@gatech.edu,orso@cc.gatech.edu

ABSTRACT

Programs typically provide a broad range of features. Because different typologies of users tend to use only a subset of these features, and unnecessary features can harm performance and security, program debloating techniques, which can reduce the size of a program by eliminating (possibly) unneeded features, are becoming increasingly popular. Most existing debloating techniques tend to focus on program-size reduction alone and, although effective, ignore other important aspects of debloating. We believe that program debloating is a multifaceted problem that must be addressed in a more general way. In this spirit, *we propose a general approach that allows for formulating program debloating as a multi-objective optimization problem.* Given a program to be debloated, our approach lets users specify (1) a usage profile for the program (i.e., a set of inputs with associated usage probabilities), (2) the factors of interest for debloating, and (3) the relative importance of these factors. Based on this information, the approach defines a suitable objective function for associating a score to every possible reduced program and aims to generate an optimal solution that maximizes the objective function. We also present and evaluate Debop, a specific instance of our approach that considers three objectives: size reduction, attack-surface reduction, and generality (i.e., the extent to which the reduced program handles inputs in the usage profile provided). Our results, albeit still preliminary, are promising and show that our approach can be effective at generating debloated programs that achieve a good trade-off between the different debloating objectives considered. Our results also provide insights on the performance of our general approach when compared to a specialized single-goal technique.

ACM Reference Format:
Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program Debloating via Stochastic Optimization. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381739

1 INTRODUCTION

Modern software systems are increasingly complex and feature-rich. Typically, however, different users use different features, and a significant fraction of these features are rarely used at all [13]. Because this unneeded functionality can severely harm performance, cost energy, and raise security issues [24], software *debloating* techniques, which can remove such unnecessary features, are becoming increasingly popular. Most existing debloating techniques (e.g., [12, 23]) take as input a (bloated) program p and a set of inputs I for p, and their only goal is to produce the smallest reduced program p' that behaves correctly (i.e., like p) for I.

We believe that this is a limited view of debloating. When performing debloating in a more realistic scenario, there would typically be different, possibly conflicting goals at play. For example, it may be acceptable for a debloated program to be unable to handle 10% of the inputs in I if that can result in an 80% reduction of its possible vulnerabilities. Similarly, there may be cases in which generality (i.e., the extent to which the reduced program can handle inputs in I) is the most important factor, and should therefore be pursued at the expense of program-size reduction. To account for these situations, and provide a more general approach to debloating, we propose a novel, general formulation of program debloating as a multi-objective optimization problem.

Our approach allows for specifying a debloating task in terms of the following elements: the program to be debloated, p; a set of inputs for p, possibly with associated usage probabilities (i.e., a usage profile); the factors of interest for the debloating task; and the relative importance of these factors, expressed as weights. Based on this information, the approach generates an objective function O that encodes the relevant factors and their weights. It then uses O to compute a score for every possible reduced program during the search for an optimal solution—the solution that maximizes O.

We also present Debop, a specific instance of our general debloating approach that supports three goals: maximizing size reduction, minimizing attack surface (code that can be leveraged for attacks), and maximizing generality. In addition to p and a user profile for p, Debop therefore takes as input weights that indicate the relative importance of the three goals and uses them to define O.

Because it is generally infeasible to enumerate all possible reduced versions of p, Debop cannot find an optimally debloated program through an exhaustive search. Instead, our technique leverages a Markov Chain Monte Carlo (MCMC) sampling technique [11] for effectively exploring the search space. The MCMC sampling algorithm, guided by the objective function O, samples a number of reduced versions of p and reports the one with the best value for O. This version, which we call p_{deb}, is an approximation of the actual optimal solution.

To evaluate our approach, we performed a case study in which we used Debop to compute debloated versions of a Unix utility using different combination of weights for its three goals. This proof-of-concept evaluation confirms that Debop can effectively generate debloated programs that achieve good trade-offs in the presence of multiple, conflicting goals. We also applied a state-of-the-art, specialized debloating technique to the same Unix utility and compared the performance of the two techniques. The result of this second study show that, although Debop may pay a price for

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381739

its generality, it produces better overall results when considering multiple goals. Moreover, DEBOP can be improved along many dimensions, as we discuss when describing our future work.

This paper makes the following contributions:

- A novel, general formulation of program debloating as a multi-objective optimization problem that accounts for the presence of multiple, possibly conflicting debloating goals.
- An instance of our approach, DEBOP, that (1) supports multiple goals expressed in terms of program size, attack surface, and generality, and (2) computes approximated solutions to the (debloating) optimization problem via MCMC sampling.
- A proof-of-concept evaluation that shows the potential usefulness of our approach and provides insight on the trade-offs involved in program debloating.
- An implementation of DEBOP that is publicly available, together with our experiment infrastructure and data, for replication, at https://sites.google.com/view/debop19.

2 BACKGROUND

Debloating. Let p be a deterministic program and I be a set of unique inputs for p with associated usage probabilities (i.e., a usage profile for p). We use $p(i)$ to denote the result of running p with input i. Given p and I, *debloating* is the process of removing code from p to produce a reduced program p'. We say that p' can *handle* an input i if $p(i) = p'(i)$.

Program Representation. A program p consists of a sequence of *program statements*: $p = \{s_1, s_2, \ldots, s_n\}$. We define $Sub(p)$ as the set of all possible reduced versions of p. We represent each program $p' \in Sub(p)$ as an n-bit bitvector $W(p') = b_1, b_2, \ldots, b_n$, where b_i is 1 when the corresponding statement s_i is present in p', and 0 otherwise. (For p, b_i is 1 for $1 \leq i \leq n$.)

3 OUR TECHNIQUE: DEBOP

As we mentioned in Section 1, our general approach formulates program debloating as a multi-objective optimization problem defined in terms of p, I, and a set of weighted goals. In this section, we present DEBOP, a specific instance of this general approach that (1) supports three goals expressed in terms of program size, attack surface, and generality and (2) computes approximated solutions to the (debloating) optimization problem via MCMC sampling.

DEBOP characterizes a reduced program $p' \in Sub(p)$ in terms of *size reduction*, $sr(p, p')$, *attack-surface reduction*, $ar(p, p')$, and *generality*, $g(p, p')$, computed as: $sr(p, p') = \frac{sz(p) - sz(p')}{sz(p)}$, where $sz(p)$ is the size of p (e.g., in LOCs); $ar(p, p') = \frac{as(p) - as(p')}{as(p)}$, where $as(p)$ measures the attack surface of p (e.g., in terms of number of gadgets [22]); $g(p, p') = \sum_{i_k \in I} pr_k T(i_k)$, where pr_k is the probability associated with the input $i_k \in I$; and $T(i_k)$ is an indicator function whose value is 1 when $p'(i_k) = p(i_k)$ and 0 otherwise. DEBOP aims to generate a reduced program p'_{deb} that maximizes size reduction, attack-surface reduction, and generality (with different weights for the different goals). To do so, it first computes a general measure of *reduction*, $r(p, p')$, by combining $sr(p, p')$ and $ar(p, p')$:

$$r(p, p') = (1 - \alpha) \cdot sr(p, p') + \alpha \cdot ar(p, p'),$$

where α is a weight between 0 and 1 that denotes the relative importance of $sr(p, p')$ and $ar(p, p')$. It then computes an objective function $O(p, p')$ defined as the weighted sum of reduction $r(p, p')$ and generality $g(p, p')$:

$$O(p, p') = (1 - \beta) \cdot r(p, p') + \beta \cdot g(p, p'),$$

where β is also a weight between 0 and 1. We call the value this function computes the *O-score*. We can now formulate DEBOP's debloating task as an optimization problem:

$$p_{deb} = \underset{p' \in Sub(p)}{\arg\max}\ O(p, p')$$

By using different values for α and β, one can generate solutions with different trade-offs between $sr(p, p')$, $ar(p, p')$, and $g(p, p')$.

Solving this optimization problem by enumerating each $p' \in Sub(p)$ and identifying the one with the highest O-score is not feasible, given the exponential size of $Sub(p)$. Therefore, DEBOP leverages stochastic search, and specifically a Markov Chain Monte-Carlo (MCMC) sampling method, to generate an approximate p_{deb}.

MCMC & Metropolis-Hastings Algorithm. MCMC is a sampling technique [11] for estimating the properties of a distribution d by drawing samples from it. To use MCMC in our context, we need to define d and its density function. We can define the density function $f(p, p')$ for d using the objective function $O(p, p')$ as

$$f(p, p') = \frac{1}{Z} exp(k \cdot O(p, p')), \tag{1}$$

where k and Z are constants [11, 20].

Our goal is to draw a sufficient number of samples S in proportion to d and infer properties of d based on S. Intuitively, this means that more samples with higher density values (and thus higher O-scores) should be drawn from d than samples with lower density values. In our context, a sample is a reduced program $p' \in Sub(p)$. To draw such samples, we use the Metropolis-Hastings (MH) algorithm [7] with the probability of accepting a new sample p'_{i+1}, given a current sample p'_i, defined as

$$\begin{aligned} A(p'_i \rightarrow p'_{i+1}) &= \min\left(1, \frac{f(p, p'_{i+1}) \cdot q(p'_i, p'_{i+1})}{f(p, p'_i) \cdot q(p'_{i+1}, p'_i)}\right) \\ &= \min\left(1, \frac{exp(k \cdot O(p, p'_{i+1})) \cdot q(p'_i, p'_{i+1})}{exp(k \cdot O(p, p'_i)) \cdot q(p'_{i+1}, p'_i)}\right), \end{aligned} \tag{2}$$

where $q(p'_i, p'_{i+1})$ is the proposal distribution of transforming one sample p'_i into another p'_{i+1}. When the transformation is symmetric (i.e., $q(p'_i, p'_{i+1}) = q(p'_{i+1}, p'_i)$), the acceptance probability becomes

$$A(p'_i \rightarrow p'_{i+1}) = \min\left(1, \frac{exp(k \cdot O(p, p'_{i+1}))}{exp(k \cdot O(p, p'_i))}\right). \tag{3}$$

The general MH algorithm starts with an initial sample p'_0. It then iteratively generates a new sample p'_{i+1} from the current one p'_i, by mutating it, and compares the density values $f(p'_{i+1})$ and $f(p'_i)$. If $f(p'_{i+1}) > f(p'_i)$, it accepts p'_{i+1} and uses p'_{i+1} as the current sample. Otherwise, it can still accept p'_{i+1} by acceptance probability $A(p'_i \rightarrow p'_{i+1})$ to avoid being trapped in local maxima. If p'_{i+1} is rejected, p'_i will be used again as the current sample.

Algorithm 1 presents the specific MH algorithm used within DEBOP. The algorithm takes as input a program p, the number of samples n to generate, and weights α and β, and generates as output a list of samples. The algorithm first obtains $stmts$, the program statements of p (line 1), sets the current sample ps as p (line 2) and

Algorithm 1 Sampling algorithm.

```
Input: p: initial program
Input: n: number of samples
Input: alpha: alpha value
Input: beta: beta value
Output: samples: samples generated
 1: stmts ← get the statements of p
 2: ps ← p
 3: dvalue ← getDensityValue(p, ps, alpha, beta)
 4: samples ← {}
 5: i ← 0
 6: while i < n do
 7:     stmt ← randomly select one statement from stmts
 8:     if ps contains stmt then
 9:         new_ps ← reduce(ps, stmt)
10:     else
11:         new_ps ← revert(ps, stmt)
12:     new_dvalue ← getDensityValue(p, new_ps, alpha, beta)
13:     accept ← false
14:     r ← getRandom()                                  ▷ r ∈ [0, 1)
15:     if r < (new_dvalue/dvalue) then
16:         accept ← true
17:     if accept then
18:         samples ← samples ∪ {new_ps}
19:         ps ← new_ps
20:         dvalue ← new_dvalue
21:         i ← i + 1
22: return samples
```

computes its density value *dvalue* (line 3), and initializes the result list *samples* and a sample counter *i* (lines 4 and 5).

The algorithm then generates samples iteratively (lines 6–21). At each iteration of the loop, it mutates the current sample *ps* to generate a new sample *new_ps*. It does so by (1) randomly selecting a program statement *stmt* from *stmts* and (2) either removing *stmt* from *ps* or adding *stmt* back to *ps* (lines 7–11). Because the transformation is symmetric, as each program statement has the same chance of being selected for removal or add-back, the algorithm uses the simplified equation 3 for computing the acceptance probability. After generating the new sample *new_ps*, the algorithm computes its density value *new_dvalue* (line 12) and compares it to the current *dvalue* to decide whether to accept *new_ps* (lines 13–16). If *new_ps* is accepted, the algorithm saves it into *samples* and updates *ps*, *dvalue*, and the sample counter *i* (lines 17–21). Finally, the algorithm returns the list of samples, among which Debop selects as p_{deb} the sample with the highest O-score.

4 PROOF-OF-CONCEPT EVALUATION

We performed a case study to investigate two research questions:

- **RQ1**: Does Debop generate debloated programs with different size reduction, attack-surface reduction, and generality trade-offs, when provided with different α and β values?
- **RQ2**: How does Debop compare with a specialized, single-goal debloating approach?

Implementation. We implemented a prototype of Debop in C++. Debop uses Clang [3] to build an AST (abstract syntax tree) for p and identify p's statements. To measure the size of a program, Debop compiles it (with GCC v. 7.4.0, -O3 option) and counts the number of bytes in the resulting executable. Then, to measure the attack surface, Debop counts the number of ROP (Return-Oriented Programming) gadgets [22] in the executable using the ROPgadget tool [5]. An ROP gadget is a sequence of machine instructions that can be exploited for ROP attacks [6].

Experiment Setup. As a benchmark for our case study, we used *mkdir* (version 5.2.1, ~28 KLOCs), a Unix utility for creating new directories used in related work [12]. As inputs (i.e., usage profile) for *mkdir*, we used the 26 tests provided with the program plus 2 tests provided with its BusyBox version [1]. (BusyBox contains a stripped-down version of several Unix utilities.) We used a timeout of 1s for each test run, to handle cases in which a reduced program did not terminate. For simplicity, we assigned equal probability to all inputs. Based on early experimentation, we configured Algorithm 1 so that it generated 1,000 samples and used $k = 50$ for computing density values.

For RQ1, we used Debop to debloat *mkdir* using all provided inputs and in multiple trials with different values for α (0.25, 0.5, and 0.75) and β (0.1, 0.2, ..., 0.9), for a total of 27 combinations. We did not include the two extreme cases of β being 0 and 1. When β is 0, Debop would simply return an empty program. Similarly, when β is 1, Debop would simply return the original program. We also did not include the two cases of α being 0 and 1, so as to consider both size reduction and attack-surface reduction when debloating. It took Debop on average 2.3 hours to finish each trial.

To the best of our knowledge, Debop is the first multi-objective debloating technique, so there are no ideal baseline techniques. Nevertheless, to get a better understanding of Debop's strengths and weaknesses, in RQ2 we compared it with a specialized, single-goal debloating technique. Specifically, we selected Chisel [12], a state-of-the-art debloating technique based on delta debugging, and used its existing implementation [2]. Chisel can only generate a reduced program that handles all the provided inputs. Therefore, to perform a fairer comparison, for each of the 27 trials, we (1) logged the exact set of inputs that Debop could correctly handle and (2) provided Chisel with the same inputs when running it within that trial. In other words, we ran Chisel on the set of "optimal" inputs that were identified as part of Debop's solution.

4.1 Results: RQ1

Figure 1 shows Debop's results for α values from 0.25 (left) to 0.75 (right). As α increases (i.e., attack-surface reduction is weighted more), the reduction score (triangle/gray line) gets closer to the attack-surface reduction score (box/orange line). Also, when β increases from 0.1 to 0.9 (i.e., generality is weighted more), the generality score increases, while the reduction score decreases. These results confirm that Debop indeed allows for exploring different trade-offs while debloating.

There are cases, however, in which generality decreases and/or reduction increases when β increases (e.g., when $\alpha = 0.25$ and β changes from 0.4 to 0.5). This could be due to the fact that some test inputs cover more program statements (i.e., some inputs are more "important"). When β is less critical (e.g., $\beta = 0.5$), omitting those test inputs but increasing the reduction has a more significant impact on improving the O-score. Also, reduced programs with less statements do not necessarily handle less inputs than reduced programs with more statements (e.g., if a failing path is added).

4.2 Results: RQ2

Table 1 shows the results of comparing Debop and Chisel in terms of size and attack-surface reduction of the debloated programs they generate. Due to space limits, we only show, for each value of α, the ratios of Debop's reduction scores over Chisel's reduction scores, presented as averages and standard deviations computed

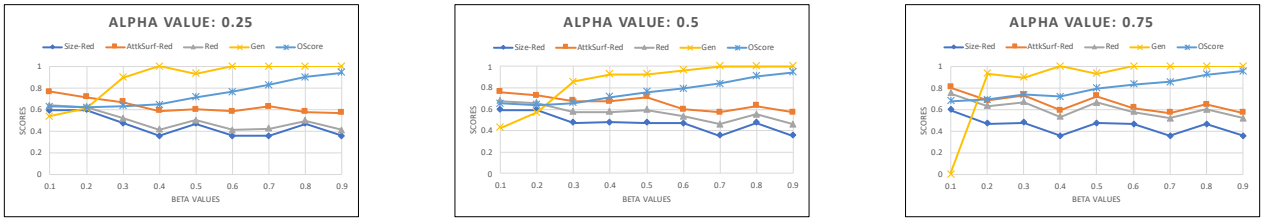

Figure 1: DEBOP's results for different values of α and β. β values are on the x-axis and the different scores are on the y-axis: Size-Red (size reduction), AttkSurf-Red (attack-surface reduction), Red (reduction), Gen (generality), and OScore (O-score).

Table 1: Average and standard deviation of the ratios of DEBOP's reduction scores over CHISEL's reduction scores for size (Size-Red) and attack surface (AttkSurf-Red).

α	Size-Red		AttkSurf-Red	
	Avg	Stdev	Avg	Stdev
0.25	0.82	0.11	1.07	0.08
0.5	0.86	0.11	1.12	0.11
0.75	0.83	0.1	1.13	0.13

over the nine β values considered. An extended version of Table 1 is available online [4].

We observe that DEBOP seems to pay a price for its generality; CHISEL, as a specialized size-reduction technique, generates debloated programs with higher size reduction than DEBOP (ratios < 1). However, by being a multi-objective technique, DEBOP can target multiple goals at once and, in fact, outperforms CHISEL in terms of attack-surface reduction (ratios > 1). It is also worth noting that, as we mentioned above, CHISEL here operates on sets of "optimal" inputs identified by DEBOP. Moreover, as we discuss in future work, DEBOP can be improved along many dimensions.

5 RELATED WORK

Many debloating techniques have been proposed in the literature [8, 12, 14, 17–19, 23]. None of them, however, formulates debloating as a multi-goal optimization problem and proposes an MCMC-based technique to solve the problem, as our work does. Our work is also related to techniques that use MCMC sampling to tackle other problems, such as superoptimization [20], and more broadly to techniques that improve software using genetic algorithms [16].

6 CONCLUSION AND FUTURE WORK

We introduced a general formulation of program debloating as a multi-objective optimization problem. We have also proposed a specific instance of this general approach, DEBOP, that considers three objectives (size reduction, attack-surface reduction, and generality) and leverages MCMC sampling to approximate an optimally debloated program. Finally, we showed the feasibility and potential usefulness of our approach through a proof-of-concept evaluation.

In future work, in addition to performing a more extensive empirical evaluation, we will investigate ways to improve Algorithm 1 through the use of (1) mutation elements other than statements and (2) hierarchical mutations based on program structure. We also plan to investigate other sampling algorithms (e.g., Gibbs sampling [10]) and other stochastic search algorithms (e.g., genetic programming [15]) for optimization. In terms of longer-term and broader future research directions, the approach we proposed is general and can be applied in areas other than debloating. In particular, we plan to investigate the use of our approach in the context of resource adaptation [8], energy reduction [21], and program repair [9].

7 ACKNOWLEDGMENTS

We thank the authors of CHISEL [12] for making their tool available. This work was partially supported by NSF, under grants CCF-1563991 and 1917924, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research.

REFERENCES

[1] 2020. *BusyBox.* https://busybox.net
[2] 2020. *Chisel.* https://github.com/aspire-project/chisel
[3] 2020. *Clang.* https://clang.llvm.org/
[4] 2020. *Extended Experimental Data.* https://docs.google.com/spreadsheets/d/12CKg-XefrfC06kDG21PKadPE8R7oVspT5IOabWTquK8/edit?usp=sharing
[5] 2020. *ROPgadget.* https://github.com/JonathanSalwan/ROPgadget
[6] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *CCS*. 27–38.
[7] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* (1995), 327–335.
[8] Arpit Christi, Alex Groce, and Rahul Gopinath. 2017. Resource adaptation via test-based software minimization. In *SASO*. 61–70.
[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *TSE* (2017), 34–67.
[10] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *TPAMI* (1984), 721–741.
[11] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. 1995. *Markov chain Monte Carlo in practice.* Chapman and Hall/CRC.
[12] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *CCS*. 380–394.
[13] Curt Hibbs, Steve Jewett, and Mike Sullivan. 2009. *The art of lean software development: a practical and incremental approach.* "O'Reilly Media, Inc.".
[14] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *COMPSAC*. 12–21.
[15] John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection.* MIT press.
[16] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2017. Genetic improvement of software: a comprehensive survey. *TEVC* (2017), 415–432.
[17] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security*. 1733–1750.
[18] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *USENIX Security*. 869–886.
[19] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *ESEC/FSE*. 476–486.
[20] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *ASPLOS*. 305–316.
[21] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler software optimization for reducing energy. In *ASPLOS*. 639–652.
[22] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *CCS*. 552–561.
[23] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *ASE*. 329–339.
[24] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER*. 421–426.