Subdomain-Based Generality-Aware Debloating

Qi Xin
Georgia Institute of Technology
qxin6@gatech.edu

Myeongsoo Kim
Georgia Institute of Technology
wardballoon@gatech.edu

Qirun Zhang
Georgia Institute of Technology
qrzhang@gatech.edu

Alessandro Orso
Georgia Institute of Technology
orso@cc.gatech.edu

ABSTRACT

Programs are becoming increasingly complex and typically contain an abundance of unneeded features, which can degrade the performance and security of the software. Recently, we have witnessed a surge of debloating techniques that aim to create a reduced version of a program by eliminating the unneeded features therein. To debloat a program, most existing techniques require a usage profile of the program, typically provided as a set of inputs I. Unfortunately, these techniques tend to generate a reduced program that is overfitted to I and thus fails to behave correctly for other inputs. To address this limitation, we propose DomGad, which has two main advantages over existing debloating approaches. First, it produces a reduced program that is guaranteed to work for subdomains, rather than for specific inputs. Second, it uses stochastic optimization to generate reduced programs that achieve a close-to-optimal tradeoff between reduction and generality (i.e., the extent to which the reduced program is able to correctly handle inputs in its whole domain). To assess the effectiveness of DomGad, we applied our approach to a benchmark of ten Unix utility programs. Our results are promising, as they show that DomGad could produce debloated programs that achieve, on average, 50% code reduction and 95% generality. Our results also show that DomGad performs well when compared with two state-of-the-art debloating approaches.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

debloating, generality-aware, stochastic optimization

ACM Reference Format:
Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Subdomain-Based Generality-Aware Debloating. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416644

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416644

1 INTRODUCTION

Today's programs are complex and provide an abundance of features [31]. Typically, however, only a small fraction of these features are commonly accessed by users [23], and the presence of unnecessary features can harm program performance, waste power, and introduce security issues [64]. For this reason, debloating techniques, which aim to remove unneeded features from a program and create a reduced version of it, are becoming increasingly popular.

Given a program P to be reduced, existing debloating techniques (e.g., [22, 41, 45, 55, 57]) usually require a *usage profile* of P, typically provided as a set of inputs I. These techniques tend to remove as much code in P as possible and generate a minimal program P_{deb} that behaves correctly for inputs in I. Because the resulting program is only guaranteed to work for I, it is likely to be overfitted to I and to fail for other inputs. We argue that a program that is guaranteed to only work for specific inputs is not generally usable, as it is rarely the case that one can provide a completely accurate usage profile.

To address this limitation of existing approaches, we propose DomGad, a novel debloating approach that has two main advantages over the state of the art. *First*, it produces reduced programs that are guaranteed to handle subdomains of inputs, rather than specific inputs; that is, DomGad produces programs that behave correctly for every possible input that belongs to these subdomains. Moreover, for any input that does not belong to a handled subdomain, the reduced programs would block the execution to avoid unexpected behaviors (e.g., crashes), so as to achieve enhanced robustness. In contrast, because reduced programs produced by an input-based approach are only guaranteed to behave correctly for specific inputs, the only way they have to avoid unexpected behaviors is to block the execution for *any* unknown input. *Second*, unlike existing approaches that take reduction as the only goal for debloating, DomGad also accounts for *generality*—the extent to which a reduced program could correctly handle inputs in its whole domain. Because there is a tension between reducing the size of a program and preserving its generality, DomGad aims to strike a balance between these two competing needs.

In our approach, we use a path π to characterize a subdomain of program P, and use the notation $\mathcal{D}(\pi)$ to indicate all the inputs of P that belong to that subdomain (i.e., all the inputs that follow the same path π). In order to produce a reduced program P' that handles a subdomain $\mathcal{D}(\pi)$, and behaves correctly for all the inputs in it, DomGad conservatively includes in P' all the code executed along path π. The overall goal of DomGad is to generate a reduced program P' that handles the set of subdomains of P that achieves the best tradeoff between reduction and generality. Intuitively, assuming the inputs are uniformly distributed across the program domain, this would correspond to producing a program that is as small as possible while being able to handle as many inputs as possible in the domain.

To achieve this goal, we formulate debloating as an optimization problem. Specifically, given a reduced program P' for P, we (i) quantify its reduction r and generality g, and (ii) define an objective function that computes an objective score based on r and g, so as to make the tradeoffs between those two values explicit. We then try to identify, among all possible reduced programs P' in the search space, the one with the highest objective score (P_{deb}).

While quantifying the reduction achieved by P' by measuring how much code has been removed from P is relatively straightforward, quantifying its generality is extremely challenging. Conceptually, one could identify every possible path π in P', and exactly count the number of inputs that follow that path using model counting [19]. Unfortunately, however, this is typically infeasible, as the number of paths within P' is generally unbounded, and model counting is complex and has conceptual limitations [19]. We therefore propose a practical technique that is based on the key insight that it is possible to model the underlying input distribution of P's domain and leverage a sampling-based approach. Specifically, DomGad (i) draws samples from the input distribution trying to identify a finite set of paths Π that can cover a significant fraction of inputs in the entire domain (i.e., it makes sense to focus on Π when debloating P) and (ii) estimates the size of the subdomain corresponding to each path $\pi \in \Pi$ based on the number of sampled inputs that result in that path. Although our sampling-based approach can only compute an approximation of the generality of a given P', it is possible to bound the error of the computed solution. Therefore, given enough samples, our approach can yield results with an estimation error being arbitrarily small.

Our overall debloating process works as follows. DomGad takes as inputs a program P and an input sampler IS that models the inputs distribution in P's domain and generates input samples. Given these inputs, DomGad performs three main steps: (1) *path identification*, (2) *path quantification*, and (3) *stochastic optimization*. In the first step, DomGad invokes IS to generate input samples and identify a finite set of paths Π that cover, with high confidence, a fraction of inputs in the domain whose combined probability is no less than a given lower bound. In the second step, DomGad invokes IS again to generate additional input samples, which it uses to estimate, for each $\pi \in \Pi$, the size of the subdomain characterized by π. Based on these estimates, DomGad computes, for any reduced program P' it generates that preserves a subset of paths of Π, the generality of P'. In this step, DomGad also computes the reduction for P', by comparing its size and attack surface (measured in terms of ROP gadgets [54]) with those of the original program P. Finally, in the third step, DomGad applies an MCMC-based approach [18] to perform stochastic optimization, with the goal of producing a debloated program P_{deb} that achieves an optimal tradeoff between reduction and generality.

To assess the usefulness of DomGad, we implemented the technique in a prototype tool and applied it to a benchmark of ten Unix utility programs used in previous work [11]. We compared DomGad with Debop [61], a generality-aware debloating technique that we developed in previous work, and Chisel [22], a state-of-the-art debloating technique that focuses exclusively on reduction. Our results are promising. DomGad was able to produce a reduced program that achieves, on average, 50% code reduction and 95% generality. Moreover, DomGad outperformed Debop in terms of generating programs with better tradeoffs between reduction and generality. Finally, DomGad was able to achieve reduction results comparable to those of Chisel, which does not consider generality.

Table 1: Paths identified for program *chown*.

Path	Input	PathProb
0	uid:sudo sf	0.146
1	-h uid:uid f	0.14
2	-h uid:uid sf	0.147
3	uid:uid d/d/d/f	0.29
4	uid:uid f	0.139
5	-R uid:sudo f	0.139

uid: user id; sf: symbolic file; f: file; d: directory.

The main contributions of this paper are:

- A new subdomain-based, generality-aware debloating technique, DomGad, that uses stochastic optimization to generate debloated programs that achieve good tradeoffs between reduction and generality.
- An empirical evaluation that shows the effectiveness of our technique and confirms that it is possible to perform generality-aware debloating.
- A prototype implementation of DomGad that is publicly available, together with our experiment infrastructure (see https://sites.google.com/view/domgad/).

2 ILLUSTRATIVE EXAMPLE

In this section we show, as an example, how DomGad debloats *chown* (v.8.2), a Unix utility that changes the user and group ownership of a file. *chown* is one of the benchmark programs [11] we used to evaluate DomGad (see Section 5.2). To apply DomGad on *chown*, we developed an input sampler based on the usage profile (i.e., set of inputs) associated with the program and provided at [11] (see Section 5.2.2 for more details).

To debloat *chown* (P), DomGad first uses the input sampler IS to identify a set of paths Π that cover, with high confidence, a fraction of inputs in the domain whose combined probability is no less than a given lower bound ($c = 0.95$), as explained in Section 4.2. This implies that the execution of P based on a random input would yield a path in Π with a 95% probability. For *chown*, a significant fraction of inputs follow a small number of paths, so the result of this step is the selection of only six paths, shown in Table 1 together with the inputs used to identify them.

In the second step (Section 4.3), DomGad uses again sampling to compute the path probability $p(\pi)$ for each π, which it uses to estimates the size of the subdomain characterized by π. For *chown*, DomGad generates a total of $N = 8321$ input samples—a number of samples that allows DomGad to have an estimation error bound within a small range (±0.03) and with a high statistical confidence (0.95). Then, for each π, DomGad counts the number n of sampled inputs that exercise π and computes $p(\pi)$ as n/N. The last column of Table 1 shows the resulting path probability for each path in our example.

A reduced program P' produced by DomGad preserves a subset of paths $\Pi' \subseteq \Pi$. DomGad computes the value of reduction r for P' in terms of size (measured as number of statements) and attack surface (measured as number of ROP gadgets [54]). Conversely, DomGad computes the value of generality g for P' as the sum of the path probabilities for all paths in Π'. As an example, the generality of a reduced program that preserves paths Nos. 1 and 3 is 0.43 (0.14 + 0.29). Given r and g, DomGad uses an objective function defined as $(1 - k_g) \cdot r + k_g \cdot g$, where $k_g \in [0, 1]$ is a weight.

In the third step, defined in Section 4.4, DomGad uses an MCMC-based approach to sample a number of reduced programs and identify P_{deb} with the highest objective score. For *chown*, using $k_g = 0.3$, DomGad produces a P_{deb} that preserves five of the six paths (all but path No. 5) and has $r = 0.63$ and $g = 0.86$; that is, P_{deb} contains 37% of the code in the original program and covers 86% of its domain, according to the distribution modeled by *IS*.

Changing the value of k_g allows DomGad to explore different tradeoffs between reduction and generality. Using $k_g = 0.7$, for instance, which gives more weight to generality, DomGad produces P_{deb} that preserves all the six paths, achieving lower reduction ($r = 0.56$) but higher generality ($g = 1$).

3 PRELIMINARY DEFINITIONS

3.1 Subdomain and Subdomain Quantification

Given a program P, its entire input domain $\mathcal{D}$, and a path π of P, we define $\mathcal{D}(\pi)$ as the subdomain of inputs that exercise π. We assume that the inputs of P in $\mathcal{D}$ follow a probability distribution with probability density function d. For an input i, $d(i) \in [0, 1]$ measures the likelihood of the occurrence of i. By definition, $\Sigma_{i \in \mathcal{D}} d(i) = 1$. We use path probability $p(\pi)$ to quantify the size of $\mathcal{D}(\pi)$, and define $p(\pi)$ as the sum of the density values for all inputs in $\mathcal{D}(\pi)$. More formally, we have

$$p(\pi) = \Sigma_{i \in \mathcal{D}(\pi)} d(i).$$

Assuming inputs in $\mathcal{D}$ are uniformly distributed, $p(\pi)$ can be simplified as $p(\pi) = \#\mathcal{D}(\pi)/\#\mathcal{D}$, where $\#\mathcal{D}(\pi)$ is the number of inputs that belong to $\mathcal{D}(\pi)$ and $\#\mathcal{D}$ is the total number of inputs.

3.2 Reduction

We measure the reduction for a program in terms of its size and attack surface. Given a program P and its reduced version P', we define the size reduction *sred* as

$$sred(P, P') = \frac{size(P) - size(P')}{size(P)},$$

where $size(\cdot)$ measures the size of a program. Similar to [22], we define $size(P)$ as the number of statements contained in P. We define the attack surface reduction *ared* as

$$ared(P, P') = \frac{attksurf(P) - attksurf(P')}{attksurf(P)},$$

where $attksurf(\cdot)$ measures the attack surface of a program. Similar to [22, 41], we define $attksurf(P)$ as the number of ROP (*Return-Oriented Programming*) gadgets [54] in P's executable. An ROP gadget is a sequence of machine instructions that ends with a return instruction and is relevant because an attacker could take advantage of a vulnerability in the program (e.g., a buffer-overflow) to overwrite a gadget's return address, hijack the control-flow, and execute malicious code [6]. Finally, we define the overall reduction *red* for a program as the weighted sum of *sred* and *ared*,

$$red(P, P') = (1 - k_r) \cdot sred(P, P') + k_r \cdot ared(P, P'),$$

where $k_r \in [0, 1]$ is the weight.

3.3 Generality

Given a reduced program P', we define generality as the measure of its ability to correctly handle inputs in $\mathcal{D}$. We say that P' can *handle* an input i if P' can produce the same output as P for i. We compute the generality *gen* for P' as the sum of path probabilities for the paths preserved in P', formally defined as

$$gen(P') = \Sigma_{\pi \in \Pi \wedge S(\pi) \subseteq S(P')} p(\pi),$$

where Π is a set of paths, $S(\pi)$ is the set of statements executed along path π, and $S(P')$ is the set of statements contained in P'. In theory, Π should include all paths of P. To make the approach practical, however, in the first step of our technique we select a finite subset of paths that cover a fraction of inputs in the domain whose combined probability is no less than a given lower bound.

3.4 Objective Function

To quantify the tradeoff between reduction *red* and generality *gen*, we define an objective function O that computes an objective score as the weighted sum of *red* and *gen*, formally defined as

$$O(P, P') = (1 - k_g) \cdot red(P, P') + k_g \cdot gen(P'),$$

where $k_g \in [0, 1]$ is the weight applied to *red* and *gen*.

3.5 Subdomain-Based Debloating

Given a program P, a set of paths Π, and the two weights k_r and k_g, the goal of subdomain-based debloating is to produce a reduced program P_{deb} that preserves a subset of paths $\Pi' \subseteq \Pi$ and maximizes O. Formally, we have

$$P_{deb} = \underset{\Pi' \subseteq \Pi}{\arg\max}\, O(P, compose(P, \Pi')),$$

where $compose(P, \Pi')$ is the reduced program that preserves all the paths in Π'. Note that one can use k_r and k_g to obtain reduced programs with different tradeoffs between *sred* and *ared* and between *red* and *gen*.

4 OUR TECHNIQUE: DOMGAD

Figure 1 provides an overview of DomGad's debloating process. We first discuss the input sampler used by DomGad and then present the three steps of the technique.

4.1 Input Sampler

DomGad relies on an input sampler to generate sampled inputs for path identification and quantification. An input sampler *IS* is a probabilistic program that uses a set of pre-defined sampling functions to generate random values. To be used within DomGad, an *IS* must provide the following four functions:

- **getUniformInt**(int *m*, int *n*): returns a random integer between *m* and *n*, selected from a uniform distribution.

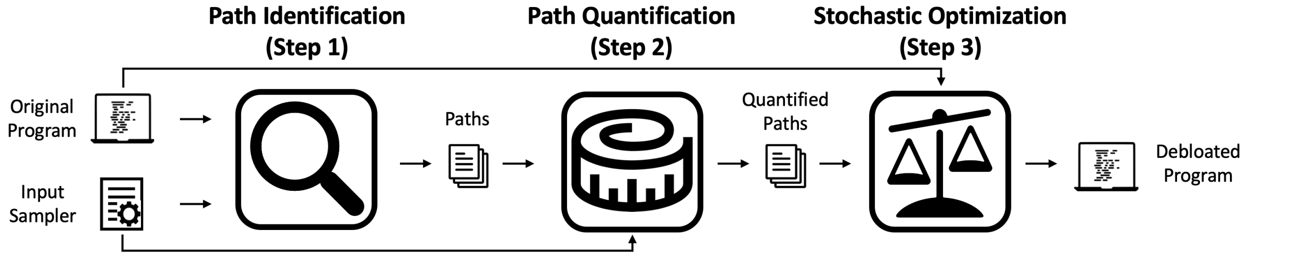

Figure 1: High-level overview of DomGad.

- **getUniformReal**(double *m*, double *n*): returns a random real number between *m* and *n*, selected from a uniform distribution.
- **getNorm**(double *mean*, double *sdev*): returns a random real number selected from a normal distribution defined by the given mean (*mean*) and standard-deviation (*sdev*).
- **getBinomial**(int *n*, double *p*): returns a random integer that represents the number of trials for which heads occur when flipping a biased coin. The total number of trials is *n*, and the bias of the coin is determined by *p*.

Although these functions do not directly produce boolean, character, or string values, it is possible to generate such values using these functions. For example, to get a random character between 'a' and 'z', we can call *getUniformInt* to generate an integer between 97 and 122 and convert it to a character.

DomGad relies on *IS* to obtain a sufficient set of sampled inputs for effective subdomain identification and quantification. In this first instance of our approach, we assume that the user is familiar with the usage of the program and can provide a reasonable sampler. As discussed in Section 7, in future work we plan to investigate automated approaches for synthesizing an input sampler, possibly based on a provided usage profile.

4.2 Path Identification

DomGad performs this step to identify a finite set of paths Π that cover, with high confidence, a fraction of inputs in P's domain whose combined probability is no less than a given domain coverage lower bound $c \in (0, 1)$. In other words, the sum of path probabilities for the paths in Π should be no smaller than c: $\Sigma_{\pi \in \Pi} p(\pi) \geq c$. To identify Π, DomGad performs a statistical, simulation-based approach that is analogous to the one used in [51] and is described in Algorithm 1.

The algorithm starts by computing parameter K, which it uses to decide when to terminate the computation, and by initializing the set of paths *PI* to the empty set (lines 1–2). It then enters its main loop (lines 4–11) and, in each iteration of the loop, it generates sample input i and computes the path *pi* exercised by i (lines 5–6). If the path is already in *PI*, the algorithm increments counter *count* (line 8). Otherwise, it resets *count* to 0 and adds *pi* to *PI* (lines 10-11). The loop terminates if no new paths are identified for K subsequent iterations.

As explained in [51], a suitable K can be computed based on parameters c and B through a Bayesian factor test [25]. Specifically, K can be computed as $K \geq \left\lceil \frac{-\log B}{\log c} \right\rceil$. By tuning c and B, DomGad could generate Π that achieves a higher coverage with higher confidence [51]. For example, given $B = 100$ and $c = 0.95$, $K \geq 90$. This means that, if DomGad does not identify new paths for 90 subsequent iterations, the resulting set of paths Π would cover, with high confidence, a set of inputs in the domain with a combined probability that is at least 0.95.

Algorithm 1 Path identification.

```
Input: P: original program
Input: IS: input sampler
Input: c: domain coverage lower bound
Input: B: confidence parameter
Output: PI: a list of paths
 1: $K \leftarrow computeK(c, B)$
 2: $PI \leftarrow \{\}$
 3: $count \leftarrow 0$
 4: while $count < K$ do
 5:     $i \leftarrow IS.getOneSample()$
 6:     $pi \leftarrow getPath(P, i)$
 7:     if $pi \in PI$ then
 8:         $count \leftarrow count + 1$
 9:     else
10:         $count \leftarrow 0$
11:         $PI \leftarrow PI \cup pi$
12: return $PI$
```

4.3 Path Quantification

In this step, DomGad performs sampling to estimate the path probability for each path in Π. For a given $\pi \in \Pi$, an input sample i drawn from the underlying distribution either (i) exercises or (ii) does not exercise π. Therefore, a sample can be considered the instance of a random variable X_i following the Bernoulli distribution $X_i \sim Bernoulli(p(\pi))$, where $p(\pi)$ is the path probability to be estimated. Let $X = X_1 + X_2 + \cdots + X_n$ be the random variable representing the sum of n independent samples (X is known to follow the Binomial distribution). We define $\hat{p}(\pi)$ as $E(X) = X/n$, the expectation of X. To compute $\hat{p}(\pi)$, the estimation of $p(\pi)$, DomGad performs a sequence of n Bernoulli trials to get n sampled inputs. Among these inputs, DomGad counts how many exercise π, n_π, and computes $\hat{p}(\pi) = n_\pi / n$.

DomGad performs acceptance sampling [50, 65] to bound errors. To do this, we define the following error-bounding constraint:

$$Pr(\hat{p}(\pi) \in [p(\pi) - \epsilon, p(\pi) + \epsilon]) \geq 1 - \alpha.$$

This constraint contains two error-bounding parameters, an accuracy parameter ϵ and a confidence parameter α, and specifies that the estimated $\hat{p}(\pi)$ will deviate from the real $p(\pi)$ by at most ϵ with probability $1 - \alpha$. By tuning ϵ and α to small values, $\hat{p}(\pi)$ gets closer to $p(\pi)$ with high confidence. Given specific ϵ and α, we use the two-sided Chernoff bound [9, 50] to compute a lower bound n_{lb} for the sampling number n as $n_{lb} = \frac{2+\epsilon}{\epsilon^2} \ln \frac{2}{\alpha}$. This means that, if

Algorithm 2 Path probability estimation.

```
Input: $P$: original program
Input: $IS$: input sampler
Input: $PI$: set of previously identified paths
Input: $\epsilon$: accuracy parameter for error-bounding
Input: $\alpha$: confidence parameter for error-bounding
Output: $pimap$: a map that maps a path to its estimated path probability
 1: $nlb \leftarrow getSampleNum(\epsilon, \alpha)$
 2: $pimap \leftarrow \{\}$
 3: for $pi \in PI$ do
 4:     $pimap.set(pi, 0)$
 5:
 6: $i \leftarrow 0$
 7: while $i <= nlb$ do
 8:     $in \leftarrow IS.getOneSample()$
 9:     $pi \leftarrow getPath(P, in)$
10:     if $pi \in PI$ then
11:         $count \leftarrow pimap.get(pi)$
12:         $pimap.set(pi, count + 1)$
13:     $i \leftarrow i + 1$
14:
15: for $pi \in PI$ do
16:     $count \leftarrow pimap.get(pi)$
17:     $pimap.set(pi, count/nlb)$
18: return $pimap$
```

we use n_{lb} samples to estimate $p(\pi)$, the estimated $\hat{p}(\pi)$ will satisfy the error-bounding constraint specified by ϵ and α.

Given program P, the input sampler IS, the set of paths PI previously identified, and the error-bounding parameters ϵ and α, DomGad uses Algorithm 2 to compute path probability $\hat{p}(pi)$ for each $pi \in PI$. The algorithm generates $pimap$, which maps each path pi to its estimated path probability. The algorithm starts by computing the number of input samples *nlb* based on ϵ and α (line 1) and initializing *pimap* by setting a key for each $pi \in PI$ and mapping it to a value 0 (lines 2–4). It then iteratively generates a total of *nlb* samples and updates *pimap* by counting the number of samples each path covers (lines 7–13). A path *pi* covers a sample *in* if running P with *in* exercises *pi*. Finally, the algorithm computes the path probability for each π based on its count and updates *pimap* (lines 15–17). Note that DomGad does not need to generate an independent set of *nlb* samples to estimate path probability for each $pi \in PI$. This is because paths are disjoint, that is, a sampled input exercises at most one path. Therefore, the random variables representing each path are independent.

4.4 Stochastic Optimization

Because there is a tension between reducing the size of a program and preserving its generality, we formulate debloating as an optimization problem. Our goal is to generate an optimally reduced program that achieves the best tradeoff between reduction and generality. Since it is generally infeasible to enumerate every reduced program in the search space, given its exponential size, DomGad performs stochastic search, using an MCMC-based approach, to find a close-to-optimal solution. We first summarize the MCMC approach we use, to make the paper self contained, and then present our stochastic optimization algorithm.

4.4.1 MCMC and Metropolis-Hastings Algorithm. An MCMC-based approach is a sampling-based approach that is commonly used for estimating properties, such as mean and variance, of a given probability distribution (whose probability density function is known). The approach performs a sequential process to draw samples from the distribution, where the generation of a new sample only depends on the previous sample.

Algorithm 3 Simplified Metropolis-Hastings algorithm.

```
Input: $f$: probability density function
Input: $N$: number of samples to be generated
Output: $S$: a set of samples
 1: $curr\_s \leftarrow$ initialize a sample
 2: $n \leftarrow 0$
 3: while $n < N$ do
 4:     $new\_s \leftarrow$ mutate $curr\_s$ by adding random noise
 5:     $ratio \leftarrow f(new\_s)/f(curr\_s)$
 6:     $rn \leftarrow$ get a uniform random number                ▷ $rn \in [0, 1)$
 7:     if $rn < ratio$ then                                   ▷ accept the new sample
 8:         $S \leftarrow S \cup new\_s$
 9:         $curr\_s \leftarrow new\_s$
10:         $n = n + 1$
11: return $S$
```

An algorithm commonly used for performing MCMC-based sampling is the *Metropolis-Hastings* (MH) algorithm, which generates new samples through mutation, by adding random noise to the current sample. A simplified version of the MH algorithm is shown as Algorithm 3. It is simplified because we assume that the mutation used for generating a new sample is *symmetric* (i.e., the probability of generating a sample s_j based on s_i is the same as that of generating s_i based on s_j). As we will show in Section 4.4.2, DomGad uses symmetric mutations to generate samples of reduced programs.

The algorithm takes as input a probability distribution defined by a probability density function f and a maximum number of samples to be generated N. It starts by initializing the current sample *curr_s* (line 1) and setting the current number of samples *n* to 0 (line 2). Next, it iteratively generates new samples (lines 3–10). In each iteration, it generates a new sample *new_s* by adding random noise to *curr_s*. To decide whether to accept *new_s* or not, it computes a density ratio *ratio* and generates a random number *rn* (lines 5–6). If *rn* is smaller than *ratio*, the algorithm accepts *new_s*. This implies that, when *new_s* is of higher density value, the algorithm always accepts *new_s*. Otherwise, when *new_s* has a lower density, it can still accept *new_s* based on its relative density drop (determined by *ratio*). Intuitively, by accepting samples this way, the algorithm is able to collect more samples from higher-density regions of the distribution, while still occasionally visiting and collecting samples from lower-density regions. This explains why the MH algorithm can generate samples that effectively approximate the given distribution. When a new sample *new_s* is accepted, the algorithm adds it to the sample set S, updates *curr_s*, and increases *n* (lines 8–10).

4.4.2 DomGad's Stochastic Approach. DomGad uses the MH algorithm to perform stochastic optimization and produce a reduced program with the highest objective score. Following existing approaches [52, 61], we define a program distribution whose probability density function f is defined based on the objective function O. Specifically, for a program P and its reduced version P', we define the probability density function $f(P, P')$ as

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

where k is a constant, and Z is the normalizing factor that ensures that the sum of density values for all programs is 1 [18, 52].

Bit-vector representation. A reduced program P' in the search space preserves a subset of paths $\Pi' \subseteq \Pi$ previously identified by DomGad. We represent this using a bit-vector. Specifically, a bit-vector $bitvec'$ for P' indicates which paths are preserved in P', where P' preserves π if it contains all the statements executed along π. A bit b' in $bitvec'$ represents a path $\pi' \in \Pi$. If b' is 1, this

```
if (x) { s0 } else { s1 }
if (y) { s2 } else { s3 }
```

```
pi_a: x->s1->y->s3
pi_b: x->s0->y->s2
pi_c: x->s1->y->s2
```

Figure 2: An example of program composition.

means that P' preserves π', and thus π' is selected to compose P'. Conversely, a value 0 for b' indicates that π' is not part of P'. It is worth noting that P' may preserve π' even if π' is not explicitly used to compose P', if the code added to preserve other paths happens to include the code for π'. Figure 2 illustrates this situation with an example: if the programs on the left preserved the two paths *pi_a* and *pi_b*, it would also "accidentally" preserve path *pi_c*. DomGad accounts for these accidentally preserved paths when computing the generality of a reduced program.

Sample mutation. DomGad mutates P' to generate a new sample P'' by randomly selecting a bit b' in P''s bit-vector and flipping it. By doing so, DomGad adds and removes entries from the set of preserved paths used to compose P''. This mutation is symmetric, as each path has the same chance of being selected (or not selected).

DomGad's algorithm. Algorithm 4 describes DomGad's stochastic optimization approach. The algorithm takes as input (1) a program P, (2) a set of previously identified paths PI, (3) a map *pimap* that maps each path $pi \in PI$ to its path probability, (4) a timeout value *timeout*, (5) *kr* and *kg*, used to compute the objective score, and (6) k, used to compute the density score. Given P and a reduced program P', we define the *density score* $d(P, P')$ as $f(P, P') \cdot Z$, which is equal to $exp(k \cdot O(P, P'))$. DomGad does not have to compute the density value and can instead use the density score to decide the acceptance of a new sample. This is because, when computing density ratio, the normalizing factor Z is a common factor and can be simplified.

The algorithm starts by generating a program with no paths (line 1), that is, a program that has an empty body for each defined function. It then computes scores for this program (lines 2–4) and initializes *currDScore*, *bestDScore*, and *bestSample*, which represent the current and highest density scores and the sample holding the highest score (lines 5–7). The algorithm also converts *PI* into a list *pi_list*, obtains its size, and creates a bit-vector *bitvec* with all bits set to 0 (lines 9–13).

The algorithm then generates samples iteratively (lines 15–48). In each iteration, it randomly flips a bit in *bitvec* (lines 16–17), computes the set S of statements executed along the preserved paths (lines 18–21), and generates a reduced program P' (i.e., a sample) accordingly (line 22). Next, the algorithm computes, for the generated program, the reduction *red* (line 24), generality *gen* (lines 25-35), and density score *dscore* (line 36) values. Note that, for computing *gen*, it would be insufficient to only consider paths explicitly selected for composing P'. As we mentioned above, the algorithm also checks for paths not selected, yet accidentally preserved in P'.

After generating a new sample, the algorithm computes the density ratio to decide whether to accept the new sample (line 39). If the sample is accepted, the algorithm updates *currDScore* and, if needed, *bestDScore* and *bestSample* (lines 42–45). Otherwise, it reverts the bit flipped (line 47). Finally, it returns *bestSample*, the sample with the highest density and objective scores (line 49).

Algorithm 4 DomGad's stochastic algorithm.

```
Input: $P$: original program
Input: $PI$: set of identified paths
Input: $pimap$: path probability map
Input: $timeout$: timeout value (in hours)
Input: $kr$: weight for computing reduction
Input: $kg$: weight for computing objective score
Input: $k$: constant for computing density value
Output: $bestSample$: resulting debloated program
 1: $P' \leftarrow$ a program with no path preserved
 2: $red \leftarrow getReductionScore(P, P', kr)$
 3: $gen \leftarrow 0$
 4: $dscore \leftarrow getDensityScore(red, gen, k, kg)$
 5: $currDScore \leftarrow dscore$
 6: $bestDScore \leftarrow dscore$
 7: $bestSample \leftarrow P'$
 8:
 9: $pi\_list \leftarrow toList(PI)$
10: $pi\_list\_size \leftarrow pi\_list.size()$
11: $bitvec \leftarrow$ new int $[pi\_list\_size]$
12: for int $i = 0; i < pi\_list\_size; i++$ do
13:     $bitvec[i] = 0$
14:
15: do
16:     $idx \leftarrow getRandomInt(0, pi\_list\_size)$        ▷ $idx \in [0, pi\_list\_size)$
17:     $bitvec[idx] \leftarrow 1 - bitvec[idx]$        ▷ Flip a bit
18:     $S \leftarrow \{\}$        ▷ A set of statements
19:     for int $i = 0; i < pi\_list\_size; i++$ do
20:         if $bitvec[i] == 1$ then
21:             $S \leftarrow S \cup getStmts(pi\_list.get(i))$
22:     $P' \leftarrow getReducedProg(P, S)$
23:
24:     $red \leftarrow getReductionScore(P, P', kr)$
25:     $gen \leftarrow 0$
26:     for int $i = 0; i < pi\_list\_size; i++$ do
27:         $preserved \leftarrow false$
28:         if $bitvec[i] == 1$ then
29:             $preserved \leftarrow true$
30:         else
31:             $S' \leftarrow getStmts(pi\_list.get(i))$
32:             if $isSubsetEqual(S', S)$ then
33:                 $preserved \leftarrow true$
34:         if $preserved$ then
35:             $gen += pimap.get(pi\_list.get(i))$
36:     $dscore \leftarrow getDensityScore(red, gen, k, kg)$
37:
38:     $accept \leftarrow false$
39:     if $random() < dscore/currDScore$ then        ▷ $random() \in [0, 1)$
40:         $accept \leftarrow true$
41:     if $accept$ then
42:         $currDScore \leftarrow dscore$
43:         if $dscore > bestDScore$ then
44:             $bestSample \leftarrow P'$
45:             $bestDScore \leftarrow dscore$
46:     else
47:         $bitvec[idx] \leftarrow 1 - bitvec[idx]$        ▷ Revert the bit
48: while $timeout$ is reached
49: return $bestSample$
```

5 EVALUATION

To assess the usefulness of DomGad, we implemented it in a prototype tool and applied it to a benchmark of ten programs. We compared DomGad to two baselines: Debop, our previous approach that also performs optimization-based debloating, and Chisel [22], a state-of-the-art, reduction-oriented technique. Specifically, we investigated four research questions:

- **RQ1**: How does DomGad perform in terms of path identification and quantification?
- **RQ2**: How does DomGad perform in terms of stochastic optimization?
- **RQ3**: How does DomGad compare with Debop in terms of the reduction-generality tradeoffs achieved by the debloated programs they generate?
- **RQ4**: How does DomGad compare with Chisel in terms of size and attack-surface reduction?

Table 2: Benchmark programs used in our evaluation.

Program	LOC	#Func	#Stmt
bzip2-1.0.5	11782	97	6154
chown-8.2	7081	122	3765
date-8.21	9695	78	4228
grep-2.19	22706	315	10977
gzip-1.2.4	8694	91	4049
mkdir-5.2.1	5056	43	1804
rm-8.4	7200	135	3835
sort-8.16	14264	233	7805
tar-1.14	30477	473	13995
uniq-8.16	7020	65	2086

5.1 Implementation Details

We developer our prototype tool using a combination of C++, Java, and Bash scripts. The tool takes as inputs a program, an input sampler, and a set of parameters (c, B, ϵ, α, $timeout$, k_r, k_g, and k), and generates a debloated program. To record paths and the statements covered in that path, our prototype uses the llvm-cov tool [37]. We relied on Clang [13] (v.9.0.0) for building the abstract syntax tree (AST) of a program and used the AST to record the starting and ending positions of the functions and statements in the program. The tool produces a reduced program based on these recorded positions and on the coverage report generated by llvm-cov. To measure the number of statements in a program, our tool leverages a utility provided by Chisel [10]. Finally, to compute attack surface reduction, the tool compiles the program using Clang and measures the number of ROP gadgets in the resulting executable using the ROPgadget tool [48].

5.2 Experiment Setup

5.2.1 Benchmark Programs. As benchmark, we used the ten Unix utility programs (the all-in-one-file versions) provided in the benchmark repository [11]. We selected these programs because they have been extensively used for evaluating debloating techniques in related work [22, 41, 61]. Table 2 shows the statistics of these programs in terms of size, number of functions, and number of statements.

5.2.2 Sampler Programs. For each benchmark program, we created an input sampler based on its usage profile, that is, based on the set of inputs associated with the program and provided in [11]. The sampler reflects how the benchmark program is used according to its usage profile. Specifically, to generate an input, the sampler randomly selects an option used in the usage profile, where an option could be an empty option, a single option (e.g., "-c"), or a combination of individual options (e.g., "-r -f"). We computed the probability of selecting an option based on its usage frequency. For example, if an option "-c" was used in seven out of the ten inputs within a usage profile, the selection probability of the option would have been 0.7.

These options, or the program in general, may require values or inputs of a specific type to operate, and the sampler must be able to provide these values and inputs. When a numeric or enumeration value is required, the sampler generates a random value from a pre-defined range of values. For example, since a permission value is needed for the "-m" option for *mkdir-5.2.1*, the sampler chooses a random value between 000 and 777. Similarly, in the case of program *date-8.21*, the sampler generates a random date or time value. When a text file is needed, the sampler produces a random file that contains N lines, where N is a random number between 1 and 100, and each line contains M ASCII characters, where M is also a random number between 1 and 100. When a compressed file is needed (for *bzip2-1.0.5*, *gzip-1.2.4*, and *tar-1.14*), the sampler generates a random text file, and then invokes the corresponding utility to generate its compressed version. Finally, if a directory is needed, the sampler generates a directory that mirrors the structure of directories in the usage profile but contains random files.

In addition, some programs require inputs with specific characteristics and relations among them. *grep*, for instance, is a Unix utility for identifying patterns within files. The sampler we developed for *grep-2.19*, does not generate a random query pattern and uses instead patterns that appear in the provided usage profile. As the target file for *grep-2.19*, the sampler first generates a random text file. Then, depending on whether the query can be found in the original input or not, the sampler will either insert the query in the target file or remove it if present.

In summary, we carefully designed the sampler programs so as to make sure they simulated how the benchmark programs are used in their usage profiles. We provide a detailed description of the sampler programs at [49].

5.2.3 Parameters. DomGad uses a set of parameters for debloating. For path identification, we set the domain coverage lower bound to $c = 0.95$, and the confidence parameter B to 100 (as suggested in [51]). With these settings, DomGad would only terminate if it does not identify any new paths for $K = 90$ subsequent iterations. We set 10000 as the maximum number of iterations for path identification. For path quantification, we set accuracy parameter ϵ to 0.03, and confidence parameter α to 0.05. With these settings, DomGad must sample $n_{lb} = 8321$ inputs to satisfy the error-bounding constraint. It is worth noting that there is a tradeoff between accuracy and efficiency for both path identification and quantification. One could decrease the value of c to sample a smaller number of inputs needed for path identification, and vice versa. Similarly, one could increase the values of ϵ and α to sample less inputs to satisfy the error-bounding constraint for path quantification, and vice versa.

With the current settings, it took DomGad 42.5 hours to finish path identification and quantification for all programs. We did not investigate how sensitive are the debloating results to the values of c, ϵ, and α, but we plan to do it in future work.

When performing stochastic optimization, we used different values of the two weights used for computing the objective score (k_r and k_g). Specifically, to study how k_g affects the debloating result, we set k_r to 0.5 and experimented with five values for k_g: 0.1, 0.3, 0.5, 0.7, and 0.9. Similarly, to study how k_r affects the results, we set k_g to 0.5 and experimented with three values for k_r: 0.25, 0.5, and 0.75. For each benchmark program, we therefore ran DomGad for a total of seven trials. In each trial, DomGad produced a debloated program, using a timeout of six hours. This resulted in a total of 420 hours of machine time to finish all trials for all programs. To compute a density score, we followed the approach used in [61] and set k to 50.

5.2.4 *Setup for Debop.*

We compared DomGad with Debop using its implementation available at [14]. Unlike DomGad, Debop is an input-based technique and requires a program and a set of inputs. To perform a fair comparison, we provided Debop with a program P_{Π} that preserves all the paths Π identified by DomGad (as DomGad would only generate reduced versions of P_{Π}). DomGad leverages a set of sampled inputs I to quantify each path in Π, and performs stochastic optimization based on the quantification result. For comparison, we provided Debop with a set of inputs $I' \subseteq I$ that only contains inputs that exercise paths in Π. We did not provide Debop with I, as there might be inputs in I that execute paths that are not in Π and are not actually used by DomGad for quantification.

For each input, Debop needs an oracle to decide whether a program P' executes correctly for that input. Therefore, for each sampler IS we developed, we also wrote a program that automatically generates an oracle for every possible input that IS generates. The oracle works by comparing the output of P' against that of its original program P. Specifically, the oracle checks a program's exit value and each output produced by the program, including files and directories. Specifically, for *bzip2-1.0.5*, *gzip-1.2.4*, and *tar-1.14*, if the program generates a text file, the oracle directly checks its content. Otherwise, if the generated file is compressed, the oracle invokes the corresponding utility to decompress it, and then checks the decompressed files. Finally, if a directory is generated, the oracle checks the files it contains. For *chown-8.4*, the oracle checks the ownership of files/directories. For *rm-8.4*, the oracle checks the existence of files/directories. For *mkdir-5.2.1*, in addition to checking the existence of the generated directories, the oracle also checks their permissions.

Debop assigns to a program that executes correctly for all the provided inputs generality 1. This is problematic (for comparison), as such a program would not be considered able to handle all inputs (in the whole domain) by DomGad. To address this problem, we slightly modified the implementation of Debop so that it takes as input a generality factor $gf \in [0, 1]$. Then, we provided Debop with a generality factor that is computed as the generality of P_{Π} (the sum of the path probabilities for all the paths in Π). In this way, for a program P', the generality score computed by Debop becomes the product of (i) gf and (ii) the number of inputs for which P' executes correctly over all provided inputs. We also configured Debop so that it quantifies reduction in the same way DomGad does.

We applied Debop to the benchmark programs using the same parameter values for k_r, k_g, and k that DomGad uses, the same number of trials, and the same timeout per trial.

5.2.5 *Setup for Chisel.*

Chisel is also input-based and thus requires a set of inputs for debloating. Similar to Debop, for each benchmark program, we provided Chisel with program P_{Π}. Because Chisel is not an optimization-based technique, we did not provide the set of inputs I' that Debop uses. Instead, we logged the exact set of paths Π'' preserved in the debloated program generated by DomGad, and obtained the set of inputs $I'' \subseteq I$ used for quantifying paths in Π''. Because I'' corresponds to the set of inputs that can be correctly handled by the programs debloated by DomGad, we provided Chisel with I''. For each input in I'', we generated an oracle using the same approach described in Section 5.2.4. For each trial performed by DomGad, we obtained the corresponding program and inputs and ran Chisel on those, using the same timeout we used for DomGad.

Table 3: Results of path identification and quantification.

Program	Path Identification			Path Quantification		
	#Paths	MaxK	Time (Hour)	#Inputs	PathProb	Time (Hour)
bzip2-1.0.5	729	90	3.2	8321	0.938	2.6
chown-8.2	6	90	<0.1	8321	1	2.6
date-8.21	401	90	1.5	8321	0.949	2.7
grep-2.19	1290	77	6.7	8321	0.935	2.9
gzip-1.2.4	373	90	1.1	8321	0.929	2.5
mkdir-5.2.1	361	90	1.3	8321	0.952	2.3
rm-8.4	252	90	0.5	8321	0.918	2.5
sort-8.16	276	90	1.2	8321	0.96	2.8
tar-1.14	20	90	0.1	8321	0.999	3.6
uniq-8.16	31	90	0.1	8321	0.991	2.3

Table 4: Reduction (Red), generality (Gen), objective score (OScore) size reduction (SizeRed), and attack surface reduction (AttkSurfRed) of the debloated programs generated by DomGad and Debop (averaged over all programs).

kr	kg	Red		Gen		OScore	
		DomGad	Debop	DomGad	Debop	DomGad	Debop
0.5	0.1	**0.82**	0.5	0.06	0.93	**0.74**	0.54
0.5	0.3	**0.7**	0.5	0.56	0.95	**0.66**	0.63
0.5	0.5	0.5	0.49	0.95	0.96	0.72	0.72
0.5	0.7	0.49	0.5	0.96	0.96	0.82	0.82
0.5	0.9	0.49	0.49	0.96	0.96	0.91	0.91

kr	kg	SizeRed		AttkSurfRed		Red	
		DomGad	Debop	DomGad	Debop	DomGad	Debop
0.25	0.5	0.67	0.67	0.31	0.31	0.58	0.58
0.5	0.5	0.67	0.67	0.33	0.31	0.5	0.49
0.75	0.5	0.67	0.67	0.32	0.32	0.41	0.4

Table 5: Size reduction (SizeRed), attack surface reduction (AttkSurfRed), and reduction (Red) of the debloated programs generated by DomGad and Chisel (averaged over all programs).

kr	kg	SizeRed		AttkSurfRed		Red	
		DomGad	Chisel	DomGad	Chisel	DomGad	Chisel
0.5	0.1	0.99	0.92	0.64	0.69	0.82	0.81
0.5	0.3	0.87	0.87	0.52	0.67	0.7	0.77
0.5	0.5	0.67	0.68	0.33	0.43	0.5	0.56
0.5	0.7	0.67	0.68	0.3	0.44	0.49	0.56
0.5	0.9	0.67	0.68	0.3	0.44	0.49	0.56
0.25	0.5	0.67	0.68	0.31	0.43	0.58	0.62
0.5	0.5	0.67	0.68	0.33	0.43	0.5	0.56
0.75	0.5	0.67	0.68	0.32	0.44	0.41	0.49

5.2.6 *Experiment Environment.*

We ran all of the experiments on a machine with a 260GB RAM, 32 AMD-Opteron 1.4GHz processors, and running Ubuntu-18.04.

5.3 Results

5.3.1 *RQ1: DomGad's performance in terms of path identification and quantification.*

Table 3 presents a summary of DomGad's path identification and quantification results. From left to right, the table shows the benchmark program (*Program*), the number of paths identified (*#Paths*), the largest K achieved during path identification (*MaxK*) (this is the largest number of subsequent iterations for which no new paths were identified), the time taken for path identification (*Time (Hour)*, 4-th column), the number of sampled inputs used for path quantification (*#Inputs*), the sum of path probabilities for the paths identified (*PathProb*), and the time taken for path quantification (*Time (Hour)*, last column).

The results show that, for all programs but *grep-2.19*, DomGad was able to identify a set of paths Π that achieved $MaxK = 90$, thus satisfying the domain coverage constraint specified by the

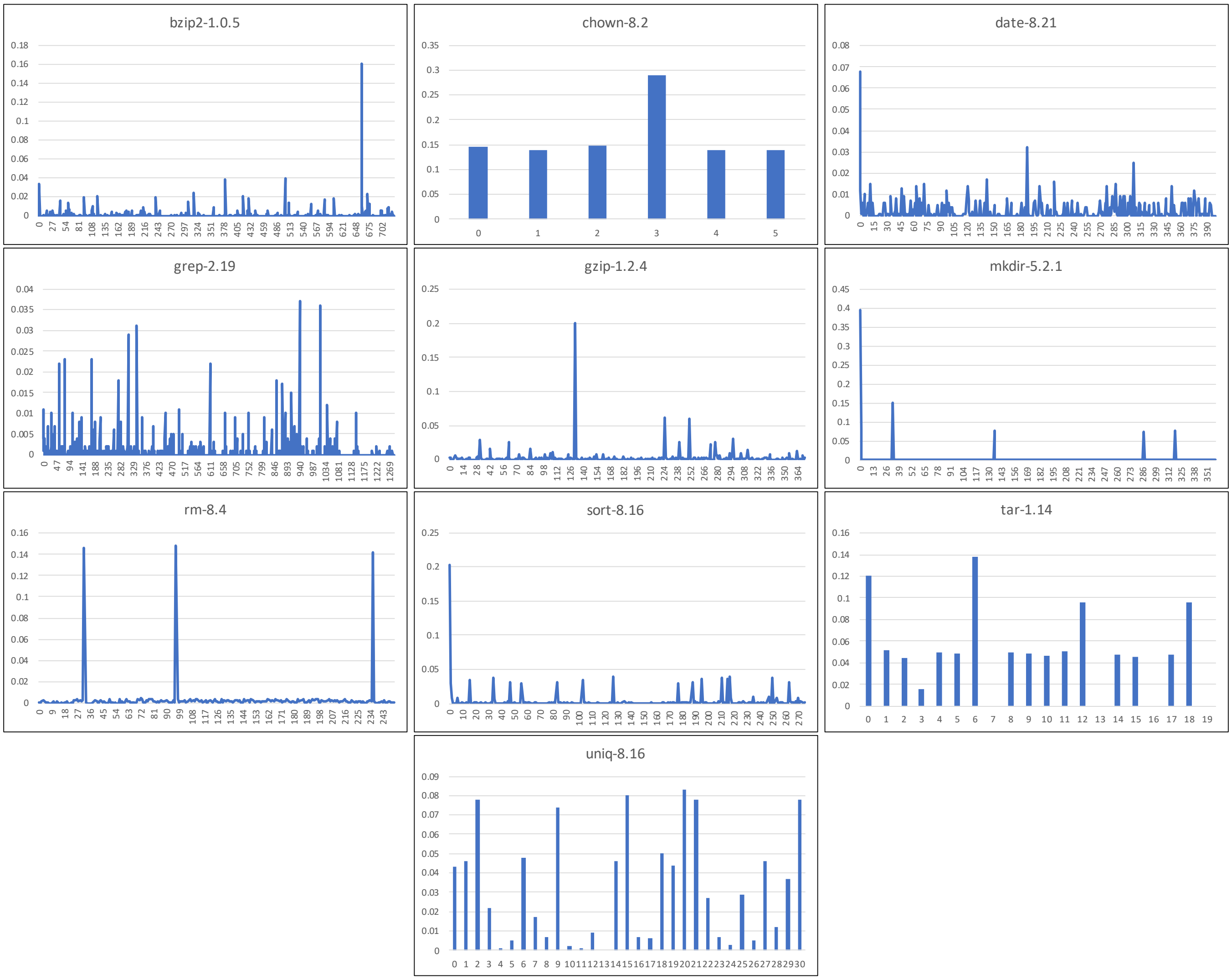

Figure 3: Path probability.

lower bound $c = 0.95$ and the confidence parameter $B = 100$. This provides initial evidence that our approach is feasible, and that it is often possible to identify a finite number of paths to achieve a high coverage of the domain (as modeled by the input sampler).

Column *PathProb* shows the sum of the probabilities for the paths in Π. For all programs, this sum is higher than 0.9. For *grep-2.19*, in particular, although set Π does not satisfy the domain coverage constraint, the estimated path probability reaches 0.935, which is only slightly lower than 0.95. For certain programs (e.g., *bzip2-1.0.5*), although Π satisfies the domain coverage constraint, the estimated probability is still lower than 0.95. This can happen, as the path probability for $\pi \in \Pi$ is estimated, and the sum could therefore be either slightly lower or slightly higher (than 0.95). Nevertheless, the average path probability over all benchmark programs is 0.957, which is fairly close to 0.95 and which indicates that DomGad’s quantification is effective.

Figure 3 presents the distribution of path probabilities for the paths identified by DomGad. For many of the programs considered, we can observe a small number of “hot paths” whose probabilities are much higher than those of other paths. This implies that it should be possible to produce, by preserving a small number of suitable paths, debloated programs that achieve good tradeoffs between reduction and generality.

5.3.2 RQ2: DomGad’s performance in terms of stochastic optimization.

Table 4 shows a summary of DomGad’s stochastic optimization results. (Full results are available at on our companion website, at https://sites.google.com/view/domgad/.) The table shows the scores of the debloated programs generated by DomGad for different k_r and k_g values, averaged over all programs. Specifically, for $k_r = 0.5$, and k_g ranging from 0.1 to 0.9, the table shows, from left to right, reduction (*Red*), generality (*Gen*), and objective score (*OScore*) for the programs generated by DomGad (and Debop). Similarly, for $k_g = 0.5$ and k_r ranging from 0.25 to 0.75, the table shows, from left to right, size reduction (*SizeRed*), attack surface reduction (*AttkSurfRed*), and reduction (*Red*) for the programs.

When k_r and k_g are both 0.5 (i.e., equal weights for size reduction and attack surface reduction, and equal weights for reduction and generality), DomGad produced a debloated program that achieves (on average) 50% reduction (67% size reduction and 33% attack surface reduction) and 95% generality. This result indicates that DomGad is able to generate, by preserving paths that achieve a high domain coverage, a reduced program that is significantly smaller (in size and attack surface), yet is highly general.

With $k_r = 0.5$ and k_g going from 0.1 to 0.9 (i.e., with increasingly higher weight given to generality and increasingly lower weight given to reduction), DomGad produced debloated programs with increasing generality (from 0.06 to 0.96) and decreasing reduction (from 0.82 to 0.49). This confirms that DomGad is indeed able to explore the space of solutions and produce debloated programs with different tradeoffs.

When $k_g = 0.5$, DomGad achieves a high generality (0.95), but when k_g changes from 0.5 to 0.9, the generality only increases slightly (0.01). When $k_g = 0.5$, we observe that, for seven benchmark programs (all but *date-8.21*, *grep-2.19*, and *gzip-1.2.4*), the debloated programs preserve all the paths identified. The reason for this result is that DomGad, in its first step, successfully identified a small set of paths Π that achieve high domain coverage. Even a reduced program that preserves all these paths is still much smaller than the original program, achieving a 0.49 reduction on average. Therefore, when reduction is not heavily weighed, DomGad tends to produce a reduced program that preserves most of the paths in Π.

When k_g is small (i.e., reduction is heavily weighed), DomGad tends to produce a debloated program that preserves only a few "hot" paths, so as to reduce code as much as possible. As an example, when $k_g = 0.1$, DomGad produced a debloated program for *mkdir* that preserves only three paths that have high probability (i.e., Nos. 0, 1, and 319 in Figure 3), achieving a reduction of 0.67 and a generality of 0.55.

Considering Table 4, we can observe that, when k_g is 0.5 and k_r ranges from 0.25 to 0.75, DomGad does not indeed produce debloated programs with different tradeoffs between size reduction and attack-surface reduction. As we previously discussed, when k_g is not extremely small, DomGad tends to produce a program that preserves all the paths, and therefore does not explore different reduction-generality tradeoffs. In future work, we will investigate how these tradeoffs vary using different values of k_g, possibly smaller than 0.5.

5.3.3 RQ3: Comparison between DomGad and Debop. Table 4 also presents Debop's results. As the table shows, Debop produced debloated programs with almost identical scores for reduction (about 0.5) and generality (about 0.95) when k_g varied from 0.1 to 0.9. In these cases, therefore, Debop failed to produce debloated programs with different tradeoffs between reduction and generality.

The reason why Debop only produced programs with high generality (even when k_g is as low as 0.1) is that its debloating process starts with a program that handles all the provided inputs, and thus preserves all the paths identified by DomGad. We observed that Debop's stochastic search is not effective at exploring the search space. The number of iterations that Debop performs (on average) for its stochastic search is 29, which is insufficient for an effective exploration.

In contrast, in the same amount of time (i.e., six hours), DomGad performed over 6000 iterations. Note that, when k_g was not extremely low (e.g., $k_g = 0.5$), Debop produced debloated programs with scores similar to those generated by DomGad. This result is due to the fact that Debop happens to start with the programs that DomGad eventually identifies as optimally reduced (i.e., the programs with all paths preserved). When k_g is low (i.e., less than 0.5), however, Debop could only generate debloated programs with lower objective scores.

We believe that there are two main reasons why Debop has limited effectiveness. First, its search space is extremely large. Debop reduces a program at the statement level, and the average number of statements in its search space is 1105, which is larger than the number of paths in DomGad's search space (374). Second, Debop, as an input-based technique, has to run the entire set of inputs to evaluate generality for every reduced program it generates, which is expensive.

It is also worth noting that we provided Debop with a reduced program that preserves all the paths identified by DomGad. Although this allows for a fair comparison between DomGad and Debop, it also makes Debop's debloating job easier, as Debop starts from the partially debloated program that DomGad generates.

5.3.4 RQ4: Comparison between DomGad and Chisel. Table 5 presents Chisel's result. As we stated above, because Chisel is a reduction-oriented technique, we provided Chisel with the exact inputs that DomGad's programs could correctly handle and only compared the two techniques in terms of reduction.

Our results show that DomGad and Chisel produce debloated programs with similar size-reduction scores, but DomGad achieves slightly higher size reduction on average. This implies that, even using an aggressive approach that focuses only on reduction, Chisel is not able to outperform DomGad and produce debloated programs with a smaller size.

In terms of attack-surface reduction, however, DomGad does not perform as well as Chisel. The reason for this is that DomGad uses a path-based approach and only eliminates statements within function bodies. Conversely, in addition to removing statements, Chisel also reduces global variables and function declarations. This helps Chisel produce a reduced program with a smaller binary size, and hence a smaller attack surface. Based on these findings, in future work we plan to investigate code-removal techniques for non-executable statements, which should improve DomGad's size-reduction performance.

It is worth noting that, since Chisel is a reduction-oriented technique, we provided it with the inputs that DomGad could correctly handle (similar to what we did for Debop). On the one hand, this allowed for a fairer comparison between Chisel and DomGad. On the other hand, however, it basically gave Chisel the advantage of operating on an already partially-debloated program.

5.4 Threats to Validity

Like all evaluations, our empirical assessment of DomGad could suffer from issues of internal and external validity. To account for possible threats to *internal validity*, we thoroughly tested and spot-checked our code. DomGad relies on an input sampler for path identification and quantification, which we developed (and which took about a day of work). To reduce bias, we designed the sampler so that they simulate how the benchmark program is used, according to its usage profile. For the two techniques we used as baseline, we leveraged the implementation provided by their authors [10, 14]. As for *threats to external validity*, we evaluated the approaches on ten Unix utility programs, and our results may not generalize to more complex programs (e.g., programs involving user interactions, database connections, and network communications) for which (1) developing effective samplers would be more challenging and (2) path identification and quantification and stochastic optimization would be more expensive and difficult. As we will discuss in Section 7, we envision a number of ways in which we could improve DomGad to address possible issues that may arise when applying it to larger and more complex benchmarks.

6 RELATED WORK

Program debloating. DomGad is related to a set of reduction-oriented techniques that rely on a usage profile for debloating [22, 41, 45, 55, 57]. TRIMMER [55] performs aggressive compiler optimization for code reduction. OCCAM [38] achieves reduction through partial evaluation [28]. C-Reduce [45], Perses [57], and Chisel [22] are reduction techniques based on delta-debugging [66]. J-Reduce [20] improves delta-debugging by leveraging dependency information for effective reduction. The reduction approach adopted by Razor [41] is based on code coverage, inference, and binary rewriting. Unlike all these techniques, DomGad performs subdomain-based debloating and produces reduced programs by focusing on subdomains, rather than specific inputs. Moreover, unlike most of these techniques, DomGad is not purely reduction-oriented; it also accounts for generality while debloating and performs stochastic optimization to strike a balance between reduction and generality. Debop [61] is a technique that we developed in previous work and that also performs optimization for debloating. Unlike DomGad, however, Debop is input-based and operates at the statement, rather than path, level. As our empirical results show, this negatively affects Debop's performance in terms of both reduction and efficiency. DomGad is also related to techniques that perform static analysis to remove dead or unused code [1, 24, 26, 27, 29, 42] and techniques that perform reduction either for specific applications (e.g., containers [44] and web applications [3]) or for special purposes (e.g., safety checking [15]). More broadly, DomGad is related to approaches for detecting bloat [4, 62, 63], identifying unnecessary code [21], and identifying code of interest through program slicing [60]. It would be interesting to investigate whether and how DomGad could be combined with some of these techniques, and in particular slicing.

Model counting and probabilistic analysis. Because DomGad performs statistical sampling for path identification and quantification, it is related to model counting techniques [2, 7, 32], which aim at quantifying the number of models that satisfy a given formula. For similar reasons, it is also related to approaches for probabilistic software analysis [5, 16, 51], which aim to quantify likelihood of the occurrence of certain probabilistic events. Finally, DomGad is related to statistical model checking techniques [35], which aim at verifying probabilistic properties through statistical methods. For path quantification, DomGad performs a hit-or-miss sampling method. Like the previous set of techniques, these approaches are mainly orthogonal to DomGad and may be interesting to investigate for identifying possible synergies.

MCMC and optimization. DomGad uses an MCMC-based approach for stochastic optimization, so it is tangentially related to techniques that leverage MCMC to tackle other problems, such as optimization [52], bug finding [8, 33], model-based GUI testing [56], and program obfuscation [36]. Finally, DomGad is loosely related to optimization techniques for resource adaptation [12], energy reduction [53], program repair [34], and, more broadly, for software improvement [40].

7 CONCLUSION AND FUTURE WORK

Existing debloating techniques are prone to producing programs that are overfitted to the specific user profile (i.e., set of inputs) used to drive the debloating process and are therefore likely to fail for most other inputs. To address this problem, we propose DomGad, a subdomain-based, generality-aware debloating technique. Unlike most existing debloating approaches, which only consider program-size reduction, DomGad also accounts for generality—a program's ability to correctly handle inputs in its whole domain. To do so, DomGad focuses on preserving specific paths, rather than individual statements, within the original program, thus producing reduced programs that are guaranteed to behave correctly for the input subdomains characterized by these paths. In order to strike a balance between reduction and generality, DomGad performs stochastic optimization using an objective function that combines these two conflicting measures and can achieve close-to-optimal tradeoffs. Our evaluation of DomGad, performed on a benchmark of ten Unix utility programs, shows that our technique can produce debloated programs that achieve significant code reductions (50% on average), while preserving high generality (95% on average). Our results also show that DomGad performs well when compared against two state-of-the-art debloating techniques.

In future work, we will first extend our evaluation by (1) applying DomGad to a broader set of programs, to assess whether our current results generalize, and (2) performing a user study, to measure the value of generality in a more realistic context. Second, we will investigate ways to improve the efficiency of path identification and quantification. In particular, we will consider approaches such as stratified sampling [47] and sequential sampling [58], as well as study the possibility of performing path identification and quantification simultaneously based on shared input samples. Third, we will consider other stochastic approaches, such as those based on Gibbs Sampling [17], to improve our optimization results. Finally, we will research ways to infer the input distribution of a program, possibly based on a usage profile, and build input samplers automatically. To do this, we will consider approaches based on probabilistic program synthesis [39], probability density estimation [59], distribution estimation [30], and deep generative models [43, 46].

8 ACKNOWLEDGMENTS

We thank the authors of Chisel [22] for making their tool available. Ravi Mangal provided helpful feedback on the approach. This work was partially supported by NSF, under grant CCF-1563991, ONR, under contract N00014-17-1-2895, and gifts from Facebook, Google, and Microsoft Research.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 70–83.

[2] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 400–410.

[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.

[4] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications (OOPSLA)*. ACM, 745–764.

[5] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S Păsăreanu, and Willem Visser. 2014. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 123–132.

[6] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*. ACM, 27–38.

[7] Supratik Chakraborty, Kuldeep S Meel, Rakesh Mistry, and Moshe Y Vardi. 2016. Approximate probabilistic inference via word-level counting. In *Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*.

[8] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41th International Conference on Software Engineering (ICSE)*. 1257–1268.

[9] Herman Chernoff. 1952. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics* (1952), 493–507.

[10] Chisel 2020. *Chisel*. https://github.com/aspire-project/chisel (accessed on September 2020).

[11] ChiselBench 2020. *ChiselBench*. https://github.com/aspire-project/chisel-bench (accessed on September 2020).

[12] Arpit Christi, Alex Groce, and Rahul Gopinath. 2017. Resource adaptation via test-based software minimization. In *11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. 61–70.

[13] Clang 2020. *Clang: a C language family frontend for LLVM*. https://clang.llvm.org/ (accessed on September 2020).

[14] Debop 2020. *Debop: Program Debloating via Stochastic Optimization*. https://sites.google.com/view/debop19 (accessed on September 2020).

[15] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-directed program trimming. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 174–185.

[16] Antonio Filieri, Corina S Păsăreanu, Willem Visser, and Jaco Geldenhuys. 2014. Statistical symbolic execution with informed sampling. In *Proc. of the ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (FSE)*. 437–448.

[17] Stuart Geman and Donald Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (1984), 721–741.

[18] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. 1995. *Markov chain Monte Carlo in practice*. Chapman and Hall/CRC.

[19] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. In *Conference on Artificial Intelligence (AAAI)*. 54–61.

[20] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 556–566.

[21] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is Static Analysis Able to Identify Unnecessary Source Code? *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020), 1–23.

[22] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 380–394.

[23] Curt Hibbs, Steve Jewett, and Mike Sullivan. 2009. *The art of lean software development: a practical and incremental approach*. "O'Reilly Media, Inc.".

[24] Jianjun Huang, Yousra Aafer, David Perry, Xiangyu Zhang, and Chen Tian. 2017. UI driven Android application reduction. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 286–296.

[25] Sumit K Jha, Edmund M Clarke, Christopher J Langmead, Axel Legay, André Platzer, and Paolo Zuliani. 2009. A bayesian approach to model checking biological systems. In *International conference on computational methods in systems biology (CMSB)*. 218–234.

[26] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 189–199.

[27] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 12–21.

[28] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[29] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*. ACM.

[30] Pedro Larrañaga and Jose A Lozano. 2001. *Estimation of distribution algorithms: A new tool for evolutionary computation*. Kluwer Academic Publishers.

[31] James R Larus. 2009. Spending Moore's dividend. *Commun. ACM* (2009), 62–69.

[32] LattE 2020. *LattE*. https://www.math.ucdavis.edu/~latte/software.php

[33] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN international conference on Object oriented programming systems languages and applications (OOPSLA)*. 386–399.

[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* (2011), 54–72.

[35] Axel Legay, Benoît Delahaye, and Saddek Bensalem. 2010. Statistical model checking: An overview. In *International conference on runtime verification*. Springer, 122–135.

[36] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 221–231.

[37] llvm-cov 2020. *llvm-cov*. https://llvm.org/docs/CommandGuide/llvm-cov.html

[38] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 1504–1511.

[39] Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. [n.d.]. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[40] 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. *IEEE Transactions on Evolutionary Computation (TEVC)* (2017), 415–432.

[41] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium (USENIX Security)*. 1733–1750.

[42] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th {USENIX} Security Symposium ({USENIX} Security)*. 869–886.

[43] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[44] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 476–486.

[45] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 335–346.

[46] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082* (2014).

[47] Christian Robert and George Casella. 2013. *Monte Carlo statistical methods*. Springer Science & Business Media.

[48] ROPgadget 2020. *ROPgadget*. https://github.com/JonathanSalwan/ROPgadget

[49] Sampler 2020. *Detailed Description of the Sampler Programs*. https://drive.google.com/open?id=1D6-RurlAOu7RMpBxXEdGNV9pCtbu8Xuh

[50] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 112–122.

[51] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. 447–458.

[52] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 305–316.

[53] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler software optimization for reducing energy.

In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*. 639–652.

[54] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).. In *ACM conference on Computer and communications security*. ACM, 552–561.

[55] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 329–339.

[56] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 25th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 245–256.

[57] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 361–371.

[58] Abraham Wald. 1945. Sequential tests of statistical hypotheses. *The annals of mathematical statistics* (1945), 117–186.

[59] Edward J Wegman. 1972. Nonparametric probability density estimation: I. A summary of available methods. *Technometrics* (1972), 533–546.

[60] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering (TSE)* (1984), 352–357.

[61] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *Proceedings of the 42st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 65–68.

[62] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 419–430.

[63] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2014).

[64] 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 *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 421–426.

[65] Håkan LS Younes. 2006. Error control for probabilistic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 142–156.

[66] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering (TSE)* (2002), 183–200.