Effective Program Debloating via Reinforcement Learning

Kihong Heo*
University of Pennsylvania, USA
kheo@cis.upenn.edu

Woosuk Lee*
University of Pennsylvania, USA
Hanyang University, Korea
woosuk@cis.upenn.edu

Pardis Pashakhanloo
University of Pennsylvania, USA
pardisp@cis.upenn.edu

Mayur Naik
University of Pennsylvania, USA
mhnaik@cis.upenn.edu

ABSTRACT

Prevalent software engineering practices such as code reuse and the "one-size-fits-all" methodology have contributed to significant and widespread increases in the size and complexity of software. The resulting software bloat has led to decreased performance and increased security vulnerabilities.

We propose a system called Chisel to enable programmers to effectively customize and debloat programs. Chisel takes as input a program to be debloated and a high-level specification of its desired functionality. The output is a reduced version of the program that is correct with respect to the specification. Chisel significantly improves upon existing program reduction systems by using a novel reinforcement learning-based approach to accelerate the search for the reduced program and scale to large programs.

Our evaluation on a suite of 10 widely used UNIX utility programs each comprising 13-90 KLOC of C source code demonstrates that Chisel is able to successfully remove all unwanted functionalities and reduce attack surfaces. Compared to two state-of-the-art program reducers C-Reduce and Perses, which time out on 6 programs and 2 programs respectively in 12 hours, Chisel runs up to 7.1x and 3.7x faster and finishes on all programs.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

program debloating; reinforcement learning

ACM Reference Format:
Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243838

*The first two authors contributed equally to this work and are listed alphabetically.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243838

1 INTRODUCTION

Software has witnessed dramatic increases in size and complexity. Prevalent software engineering practices are a key factor behind this trend. For instance, these practices emphasize increasing developers' productivity through code reuse. Moreover, they espouse a "one-size-fits-all" methodology whereby many software features are packaged into a reusable code module (e.g., a library) designed to support diverse clients.

The resulting software bloat has led to decreased performance and increased security vulnerabilities. Moreover, the abundance of gadgets in common libraries allows attackers to execute arbitrary algorithms without injecting any new code into the application.

Although software often contains extraneous features that are seldom if ever used by average users, they are commonly included without providing users with any practical or effective means to disable or remove those features. The prevailing approach is to reimplement lightweight counterparts of existing programs. Examples include web servers [10], databases [14], C/C++ libraries [7, 16], and command-line utilities [3, 15], all of which were reimplemented to target embedded platforms. However, this approach is only applicable when source code is available, and requires significant manual effort. In the context of mobile applications, app thinning [2] was recently introduced for iOS apps to automatically detect the user's device type and only download relevant content for the specific device. While a promising step towards addressing bloat, it requires developers to tag their software to identify correspondences, which has led to its sparing use even on iOS [2].

We set out to develop a practical system to enable programmers to customize and debloat programs. The system takes as input a program to be simplified and a high-level specification of its desired functionality, and generates a minimized version of the program that is correct with respect to the specification. We identified five key criteria that such a system must satisfy to be effective:

- *Minimality*: Does the system trim code as aggressively as possible while respecting the specification?
- *Efficiency*: Does the system efficiently find the minimized program and does it scale to large programs?
- *Robustness*: Does the system avoid introducing new errors and vulnerabilities in the generated program?
- *Naturalness*: Does the system produce debloated code that is maintainable and extensible?
- *Generality*: Does the system handle a wide variety of different kinds of programs and specifications?

In this paper, we present a software debloating system named Chisel that satisfies the above criteria. As depicted in Figure 1, Chisel takes a program P to be minimized and a property test function S that checks if a candidate program satisfies or violates the property. The output is a minimized version P' of the program that satisfies the property.

Chisel provides a formal guarantee on the minimality of the generated program, called *1-minimality* [45], which has been shown to suffice in the literature on program reduction [32, 36, 37]. The property test function can be expensive to invoke; for instance, it may involve compiling the candidate program and running it on a test suite. The 1-minimality guarantee admits minimization algorithms that in the worst case invoke the property test function a quadratic number of times in the size of the input program. However, even with this lesser guarantee than *global minimality*—which has worst-case exponential behavior—it is challenging to scale to large programs. Chisel overcomes this problem by avoiding generating a large number of syntactically or semantically invalid candidate programs during its search.

Chisel guarantees that the minimized program is correct with respect to the given property and is therefore robust. It avoids program transformations that could mangle the program or break its naturalness [24]. Finally, it treats both the program and the property as black-boxes, enabling it to be applicable to a wide range of different kinds of programs and specifications.

On the other hand, state-of-the-art program reduction tools such as C-Reduce [36] and Perses [37] do not satisfy all of the above criteria. Like Chisel, both of these tools take a program to be minimized and an arbitrary property test function, and return a minimized version of the program. While C-Reduce satisfies the same minimality and correctness criteria as Chisel, however, it sacrifices efficiency, naturalness, and generality. C-Reduce is tightly coupled with hand-crafted program transformation rules that are tailored to C/C++. Since the rules are myopic, C-Reduce generates a significant number of *syntactically* invalid candidates during its search for a minimal version of the given program. Moreover, the tool often generates unnatural code (see Section 5).

Perses also sacrifices efficiency and generality. Its reduction process is syntax-guided, which enables it to overcome a limitation of C-Reduce by avoiding generating syntactically invalid programs during its search. However, the tool still suffers from limited scalability by generating a large number of *semantically* invalid programs during its search. The algorithm is unaware of semantic dependencies between program elements (e.g., def-use relations of variables). As a result, it often generates programs with semantic errors, such as uninitialized variables. Also, the grammar-aware reduction can be overly conservative in each reduction step and thereby less efficient than C-Reduce. Lastly, Perses is not applicable when even correct parsing is not feasible (e.g., for binary programs).

Our main technical insight to overcome the above limitations of existing program reduction techniques is to accelerate program reduction via *reinforcement learning* [38]. From repeated trial and error, Chisel builds and refines a statistical model that determines the likelihood of each candidate program's passing the property test. The model effectively captures semantic dependencies between program elements and guides the search towards a desirable minimal program. The learning method employed by Chisel is agnostic

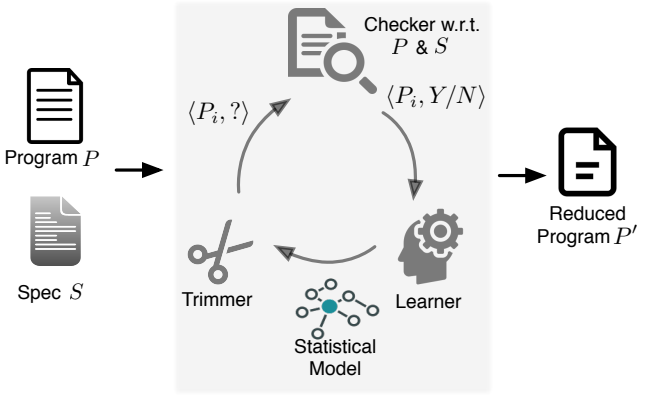


Figure 1: Overview of the Chisel system.

of the targeted programming language and specification since the models are learned from simple vector representations of tried candidate programs and their property test results.

We evaluate Chisel on a suite of 10 widely used UNIX utility programs each comprising 13-90 KLOC of C source code. Chisel efficiently converges to the desired minimal programs and outperforms existing program reduction tools. Compared to C-Reduce and Perses, which time out on 6 programs and 2 programs respectively in 12 hours, Chisel achieves up to 7.1x and 3.7x speedup and finishes on all programs. It successfully trims out 6 known vulnerabilities (CVEs) in 10 programs and eliminates 66.2% of the available gadgets on average. The robustness of the debloated programs is further validated by running a state-of-the-art fuzzer AFL [1] for three days. Furthermore, we also manually analyze the source code of the generated program to confirm that any removed functionality is as intended, and that desirable software engineering practices such as modularity and locality are preserved.

In summary, this paper makes the following contributions.

- We propose a practical system Chisel to reduce the size and complexity of software. It aims to remove unwanted functionalities from existing programs to reduce their attack surfaces.
- We propose a general reinforcement learning framework for efficient and scalable program reduction. Our algorithm is agnostic of the targeted programming language and specification.
- We evaluate Chisel using a set of widely used UNIX utility programs. Our experiments demonstrate that it enables removing existing known vulnerabilities and reducing attack surfaces without introducing any new bugs.

2 MOTIVATING EXAMPLE

We illustrate how Chisel enables programmers to customize and debloat programs using the example of a UNIX utility called `tar`. Suppose we want to obtain a simplified version of `tar` to target embedded platforms. There exists such a version of `tar` in a UNIX utility package for embedded Linux called BusyBox [3]. The original `tar` provides 97 command-line options, whereas its lightweight counterpart in BusyBox only provides 8 options. We demonstrate how to automatically obtain a program that has the same functionality as the BusyBox version by providing a simple and high-level

```
1  #!/bin/bash
2
3  function compile {
4    clang −o tar.debloat tar−1.14.c
5    return $?
6  }
7
8  function core {
9    # test 1: archiving multiple files
10   touch foo bar
11   ./tar.debloat cf foo.tar foo bar
12   rm foo bar
13   ./tar.debloat xf foo.tar
14   test −f foo −a −f bar || exit 1
15
16   # test 2: extracting from stdin
17   touch foo
18   ./tar.debloat cf foo.tar foo
19   rm foo
20   cat foo.tar | ./tar.debloat x
21   test −f foo || exit 1
22   ... #12 more tests that exercise the 8 target options
23   return 0
24 }
25
26 function non_core {
27   for test_script in `ls other_tests/*.sh`; do # for all optional test cases
28     { sh −x −e $test_script; } >& log
29     grep 'Segmentation fault' log && exit 1
30   done
31   return 0
32 }
33
34 compile || exit 1
35 core || exit 1
36 non_core || exit 1
```

Figure 2: Example test script for debloating `tar`.

specification to Chisel. We also show how such debloating can lead to concise code and enhanced security. Lastly, we explain how we can guarantee the robustness of the resulting program.

2.1 Specifying the Inputs to Chisel

First of all, we require the user to write a high-level specification to describe desired features. Such a specification can be written as a script program that takes a program source, compiles it, and checks if input-output behaviors of the compiled program are desirable. If any errors or inconsistencies are found during the steps, the script program returns false; otherwise, it returns true.

Figure 2 depicts such a script program that can be used as a specification. Recall that our goal is to obtain a program with the same functionality as `tar` in BusyBox that provides core functionalities of `tar` with 8 command-line options. The script comprises three steps. In the first step, the first function `compile` is invoked to check if a given source is compilable. If the program can be compiled, the next function `core` checks if the program exhibits the desired property. This step comprises 14 test cases that exercise the 8 command-line options. For example, the first test case tries to archive two files and extract files from the archive, and checks if the correct files are extracted (line 14). The function returns success only if the program passes all of the 14 tests. The last function `non_core` serves to avoid introducing new errors by debloating. It specifies the condition that the reduced program should at least not crash on inputs that exercise undesired features (line 29). Without this requirement, the reducer may arbitrarily remove code parts for non-core functionalities, possibly making the reduced program exploitable when a removed feature is invoked.

To write such a script, we need test cases that extensively exercise the whole functionalities of the target programs. Such test cases can be obtained in various ways; automatic test generation techniques or regression test suites by developers can be used. In this example, we used the test suites written by developers of the original program.

2.2 Result of Chisel vs. Other Approaches

Given the specification and the original version of `tar` comprising 45,778 LOC (13,227 statements), within 12 hours, Chisel generates a simplified version comprising only 1,687 LOC (538 statements), which is about 10% of the original size. The guided search by the learned statistical model, detailed in Section 4, enables to efficiently find a minimal solution. In contrast, the state-of-the-art program reducers Perses [37] and C-Reduce [36] fail to find a minimal program within 12 hours.

Figures 3 and 4 depict the reduced and the original versions of `tar`, respectively. In the `main` function of the reduced version, the code for handling command-line options has been simplified compared to the original version since the reduced program supports fewer options. Besides `main`, two functions `read_and` and `safer_name_suffix` are also simplified. In the original version, the function `read_and` checks the header of a given input file and provides an exception handling mechanism if the header is in an invalid form. If the header is valid, a function is invoked according to the given command-line options. In the reduced version, the exception handling part is removed, and the program quietly terminates if the header is invalid. Moreover, in the reduced version, function `safer_name_suffix` is also significantly simplified by removing the redundant branch.

Achieving the above program reduction is not straightforward by using typical static or dynamic analyses alone. Static analysis conservatively includes all the code parts since the actual command-line options and the input file are unknown at compile time. Thus, the static approach cannot remove any code in function `read_and` because the value of variable `status` at line 29 is unknown. On the other hand, dynamic reachability cannot be used to remove any code in function `safer_name_suffix`. Because our test cases do not exercise option '-P', variable `absolute_name` is always set to zero. As a result, the dynamic approach always covers the else branch at line 9, and so it cannot remove the else branch starting from line 9 that contains a security vulnerability discussed below.

2.3 Analyzing the Output of Chisel

We now describe two salient aspects of the reduced version of `tar` generated by Chisel: removing a security vulnerability and facilitating further validations.

```
1  /* CHISEL: global variable declarations removed */
2  struct tar_stat_info stat_info;
3
4  char *safer_name_suffix (char *file_name, int link_target) {
5      /* CHISEL: code containing CVE removed */
6      return file_name;
7  }
8
9  void extract_archive() {
10     char *file_name = safer_name_suffix(stat_info.file_name, 0);
11     /* CHISEL: overwriting functionalities removed */
12 }
13
14 void list_archive() { ... /* same as original */ }
15
16 void read_and(void *(do_something)(void)) {
17     enum read_header status;
18     while (...) {
19         status = read_header();
20         switch (status) {
21         case HEADER_SUCCESS: (*do_something)(); continue;
22         /* CHISEL: unnecessary functionalities removed */
23         default: break;
24         }
25     }
26 }
27
28 /* Supports only 8 options: -c, -f, -x, -v, -t, -O, -o, -k */
29 int main(int argc, char **argv) {
30     int optchar;
31     while (optchar = getopt_long(argc, argv) != -1) {
32         switch (optchar) {
33         case 'x': read_and(&extract_archive); break;
34         case 't': read_and(&list_archive); break;
35         /* CHISEL: unsupported options removed */
36         }
37     }
38     ... /* same as original */
39 }
```

Figure 3: Code snippet of debloated version of `tar` generated by CHISEL.

Security Vulnerability Removal. When `tar` extracts files from an archive, by default it writes into files relative to the working directory. That is because if an archive were generated by an untrusted user and contained malicious content with absolute paths, that user could potentially write into any file owned by the one who extracts from the archive.

However, handling relative paths poses a challenge when an archive contains '`../`' in its target pathname. Malicious content can be written to any file by escaping the intended destination through '`../`'. To avoid this problem, `tar` 1.14 through 1.29 have a sanitization mechanism implemented in the function `safer_name_suffix` shown in Figure 4. The sanitizer changes a given pathname by removing its prefix containing '`../`', i.e., the new pathname is the longest suffix without '`../`'. For example, pathname '`a/../b`' becomes '`b`' after sanitization.

Unfortunately, this sanitization is also flawed. A recently discovered CVE [5] exploits a problem in sanitizing such file names. If a target system is extracting an attacker supplied file, the vulnerability allows the attacker to gain file overwrite capability. A realistic attack scenario [8] is as follows. Suppose a root user wants to unknowingly download an archive file from a malicious server and extract a message-of-the-day [1] file in the archive. The malicious archive `tar-poc.tar` contains an entry whose member name is '`etc/motd/../etc/shadow`'. By typing the following command in the root (`/`) directory, the root user intends to only extract a file from the downloaded archive and write to '`/etc/motd`'.

```
$ tar xf tar-poc.tar etc/motd
```

By exploiting the vulnerability, the file '`/etc/shadow`' is changed. Note that '`/etc/shadow`' should not be extracted when asking for '`/etc/motd`'. The '`/etc/shadow`' file stores actual passwords in encrypted format for users' (including the root) accounts. Thus, in the worst-case scenario, this vulnerability can lead to a full system compromise.

The developers of `tar` fixed this issue by simply ignoring entries whose pathname contains '`../`' [2]. Thus, subsequent versions of `tar` skip extracting the entry '`etc/motd/../etc/shadow`', and the CVE is trivially disabled. On the other hand, this version also disallows any benign use of '`../`' that does not overwrite existing files.

In the reduced version by CHISEL, this exploit is not reproducible, albeit due to a different reason. CHISEL removes the feature of overwriting existing files (line 21) as well as the wrong sanitization (line 10) since those features are not exercised as core functionalities by the test script. As a result, in our version, the input archive files can be extracted to the outside of the current working directory with '`../`', but existing files are not overwritten with malicious content. Hence, the exploit is not possible in the reduced version.

Note that this reduced version cannot be obtained by simply removing code not covered by the executions using the test-cases. Since the variable `absolute_names` always holds 0 in our test cases, only the else branch is taken. In other words, if we debloated the program with a dynamic reachability-based method, the attacker could still gain file overwrite capability.

Scope for Further Validation. Since the reduced programs are typically very small, yet largely preserve their syntactic structures, the user can easily verify the differences. Sophisticated code comparison tools can clearly show the reduction like the above examples in Figures 3 and 4.

Furthermore, the reduced size and complexity also enable to apply precise automated techniques. To check if the reduced version introduces new bugs, existing program reasoning techniques (e.g., static/dynamic analysis, fuzzing, runtime monitoring mechanisms, or verification) can be employed to improve correctness guarantees. We analyzed the reduced `tar` program using static analysis and random fuzzing.

Debloating makes it feasible to manually inspect the results of static analysis. For example, Sparrow [13]—a static analyzer for finding security bugs—generated only 19 alarms for the reduced `tar` program within a second, as opposed to 1,290 alarms for the

[1] The file '/etc/motd' on Unix systems contains a "message of the day" and is used to send a common message to all users.
[2] http://git.savannah.gnu.org/cgit/tar.git/commit/?id=7340f67b9860ea0531c1450e5aa261c50f67165d

```
1  int absolute_names;
2  int ignore_zeros_option;
3  struct tar_stat_info stat_info;
4
5  char *safer_name_suffix (char *file_name, int link_target) {
6      char *p;
7      if (absolute_names) {
8          p = file_name;
9      } else {
10         /* CVE-2016-6321 */
11         /* Incorrect sanitization when "file_name" contains ".." */
12         /* "p" points to the longest suffix of "file_name" without "../" */
13         ...
14     }
15     ...
16     return p;
17 }
18
19 void extract_archive() {
20     char *file_name = safer_name_suffix(stat_info.file_name, 0);
21     /* Overwrite "file_name" if exists */
22     ...
23 }
24
25 void list_archive() { ... }
26 void read_and(void *(do_something)(void)) {
27     while (...) {
28         enum read_header status = read_header();
29         switch (status) {
30         case HEADER_SUCCESS: (*do_something)(); continue;
31         case HEADER_ZERO_BLOCK:
32           if (ignore_zeros_option) continue;
33           else break;
34         ...
35         default: break;
36         }
37     }
38     ...
39 }
40
41 /* Support all options: -x, -t, -P, -i, ... */
42 int main(int argc, char **argv) {
43     int optchar;
44     while (optchar = getopt_long(argc, argv) != -1) {
45         switch (optchar) {
46         case 'x': read_and(&extract_archive); break;
47         case 't': read_and(&list_archive); break;
48         case 'P': absolute_names = 1; break;
49         case 'i': ignore_zeros_option = 1; break;
50         ...
51         }
52     }
53     ...
54 }
```

Figure 4: Code snippet of the original version of `tar`.

original. After manual inspection, we concluded that all of the 19 alarms are false.

The debloated program can also be efficiently tested by random testing tools like fuzzers. We ran the AFL tool [1] on the reduced `tar` program and it did not find any failure-inducing inputs even within three days. This provides improved confidence in the correctness of the debloated program.

3 BACKGROUND

This section formalizes our setting for program debloating. It also introduces the concepts of delta debugging and reinforcement learning upon which our program debloating approach is based.

3.1 Program Debloating

Let $P \in \mathbb{P}$ be a program where $\mathbb{P}$ is the universe of all possible programs. A *property* is described as a property test function $O : \mathbb{P} \rightarrow \mathbb{B}$ where $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ such that $O(P) = \mathsf{T}$ if P exhibits the property, otherwise $O(P) = \mathsf{F}$. Let $|P|$ denote the size of P according to an appropriate metric such as statements or tokens.

Given a program P and a property test function O such that $O(P) = \mathsf{T}$, the goal of program debloating is to search for a minimized program $P^* \in \mathbb{P}$:

$$P^* = \arg\min_{P' \in \mathbb{P}} |P'| \quad s.t. \quad O(P') = \mathsf{T}.$$

Achieving this goal, called *global minimality*, is NP-complete [45]. Therefore, the debloating problem in practice is relaxed to target a more feasible goal called *1-minimality* [45]. A program $P^* \in \mathbb{P}$ is called 1-minimal if any variant P' derived from P^* by removing a single element from P^* does not pass the property test.

3.2 Delta Debugging

We next briefly introduce a program debloating algorithm using Delta Debugging (DD for short) [45] in Algorithm 1. Given an input program P and a property O, DD first converts the input program into a list L of elements of arbitrary granularity such as tokens, lines, or functions (line 1). The initial solution candidate and the number of partitions n are set to L and 2, respectively (line 1 and 2). The current solution candidate L is split into n partitions (line 4). For each partition u_i, the algorithm tests if the partition (resp., its complement) can preserve the property (lines 5 and 7). If so, it removes the complement of u_i (resp., u_i) from L, and resumes the main loop (lines 6 and 8). If a partition passes the property test, DD repeats the process with the coarsest granularity by setting $n \leftarrow 2$. If a complement passes the test, DD maintains the current level of granularity by setting $n \leftarrow n - 1$. When none of the partitions and

Algorithm 1 Delta Debugging

Input: A program P
Input: A property test function O
Output: A minimal program P' such that $O(P') = \top$

1: $L \leftarrow$ A list representation of P
2: $n \leftarrow 2$
3: **repeat**
4: $\quad \langle u_1, \ldots, u_n \rangle \leftarrow$ split L into n partitions
5: $\quad$ **if** $\exists i.\ O(u_i) = \top$ **then**
6: $\quad\quad \langle L, n \rangle \leftarrow \langle u_i, 2 \rangle$
7: $\quad$ **else if** $\exists i.\ O(L \setminus u_i) = \top$ **then**
8: $\quad\quad \langle L, n \rangle \leftarrow \langle L \setminus u_i, n - 1 \rangle$
9: $\quad$ **else**
10: $\quad\quad \langle L, n \rangle \leftarrow \langle L, 2n \rangle$
11: $\quad$ **end if**
12: **until** $n \leq |L|$
13: **return** P' corresponding to L

```
1  int f1 () { return 0; }
2  int f2 () { return 1; }
3  int f3 () { return 1; }
4  int f4 () { return 1; }
5  int f5 () { return 1; }
6  int f6 () { return 1; }
7  int f7 () { return 1; }
8  int main () { return f1(); }
```

Figure 5: Example program.

their complements satisfy the property, DD tries to split each partition into halves. If each partition cannot be split (line 12), it returns the program P' corresponding to the list of remaining elements L (line 13). Otherwise, it resumes the main loop by splitting each remaining partition into halves (line 10). The worst-case complexity of this algorithm is $O(|P|^2)$.

We next illustrate the DD algorithm on an example which we use in the rest of the paper as the running example.

Example 3.1. Consider the following simple C code in Figure 5. Suppose the desired property is a process termination with 0, and we are reducing the program with the granularity of function definitions. Therefore, the bare minimum is the program that only contains function `f1` and `main`. Although the minimal solution can be obtained through a simple static analysis, we depict how the DD algorithm works presuming a general setting where such an analysis may not be available.

Figure 6 depicts iterations of the algorithm. In the first two iterations, the algorithm tries two partitions ($n = 2$), each of which comprises the first four and the last four lines, respectively. Since both partitions fail to preserve the property, the algorithm increases the granularity by setting $n = 4$, and tries the four partitions, all of which fail (iterations 3–6). The algorithm then tries complements of the four partitions. In the 8th iteration, the algorithm finds a complement that preserves the property. By decrementing n by 1, the algorithm maintains the current granularity and tries $n = 3$ subsets of the current candidate. Since all of the three subsets

	f1	f2	f3	f4	f5	f6	f7	main	✔
1	f1	f2	f3	f4	f5	f6	f7	main	✘
2	f1	f2	f3	f4	f5	f6	f7	main	✘
3	f1	f2	f3	f4	f5	f6	f7	main	✘
4	f1	f2	f3	f4	f5	f6	f7	main	✘
5	f1	f2	f3	f4	f5	f6	f7	main	✘
6	f1	f2	f3	f4	f5	f6	f7	main	✘
7	f1	f2	f3	f4	f5	f6	f7	main	✘
8	f1	f2	f3	f4	f5	f6	f7	main	✔
9	f1	f2	f3	f4	f5	f6	f7	main	✔
10	f1	f2	f3	f4	f5	f6	f7	main	✘
11	f1	f2	f3	f4	f5	f6	f7	main	✘
12	f1	f2	f3	f4	f5	f6	f7	main	✘
13	f1	f2	f3	f4	f5	f6	f7	main	✘
14	f1	f2	f3	f4	f5	f6	f7	main	✘
15	f1	f2	f3	f4	f5	f6	f7	main	✔
16	f1	f2	f3	f4	f5	f6	f7	main	✔

Figure 6: DD iterations.

($\langle$`f1`, `f2`$\rangle$, $\langle$`f5`, `f6`$\rangle$, $\langle$`f7`, `main`$\rangle$) were already tried, they are skipped. Then it tries their complements, and another smaller program is found in the 9th iteration. By decrementing n by 1 again, the algorithm maintains the current granularity and tries $n = 2$ subsets of the current candidate. Again, all of the two subsets (and their complements) were already tried and failed. Now the algorithm doubles the granularity ($n \leftarrow 2 \times 2$) and tries four subsets (iterations 10–13), all of which fail. Proceeding to their complements, in the 15th iteration, another correct complement is found. Now it tries $n = 3$ subsets and their complements of the program, and the minimal solution is found in the last iteration.

3.3 Reinforcement Learning

Markov Decision Process. Markov decision process (MDP) is a framework for sequential decision making problems [38]. An *agent* is the learner and decision maker who interacts with the so called *environment*. The agent gets a reward from the environment depending on *actions* at each *state*. Formally, a MDP comprises the following components:

- A set of states $\mathcal{S}$ whose initial state is denoted as $s_0 \in \mathcal{S}$.
- A set of actions $\mathcal{A}$ and function $A : \mathcal{S} \rightarrow 2^{\mathcal{A}}$ specifying available actions at each state.
- The transition model $T : \mathcal{S} \times \mathcal{A} \rightarrow Pr(\mathcal{S})$ where $T(s' \mid s, a)$ denotes the probability of transition to state s' from state s taking action a.
- The reward function $R : \mathcal{S} \rightarrow \mathbb{R}$ where $R(s)$ denotes the expected reward at a state to s.

Solving MDP is to find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that specifies a desirable action that an agent takes in a given state. Usually, we are interested in finding an *optimal policy* π^* defined for each state $s \in \mathcal{S}$ as follows:

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} T(s' \mid s, a) V^*(s')$$

where $V^*(s)$ is the expected sum of rewards if the agent executes an optimal policy starting in state s, which is recursively defined

as follows:

$$V^*(s) = R(s) + \gamma \sum_{s'} T(s' \mid s, \pi^*(s)) V^*(s') \quad (0 \leq \gamma < 1)$$

where γ is a discount rate that determines the present value of future rewards. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Model-based Reinforcement Learning. Model-based reinforcement learning (MBRL) is a technique to solve MDP under the guidance of a *model* that predicts how the environment responds to the agent's action at each state [38]. MBRL learns such a model for transition probabilities and rewards while solving of MDP. MBRL keeps track of state transitions and actions to update the model based on the obtained information. For each state, the agent estimates the expected sum of future rewards based on the model and the action that maximizes the expected sum of future rewards.

4 OUR APPROACH

This section presents our learning-based program debloating approach. We start by illustrating our algorithm on the running example. We then present the algorithm by instantiating MDP with delta debugging. Lastly, we explain how a statistical model is learned during the delta debugging process.

4.1 Informal Description

Our key insight is to aim to quickly converge to a solution by prioritizing candidates that are likely to pass the property test. Recall that 11 of the 16 trials made by the DD algorithm in Example 3.1 failed the property test. We aim to avoid such a high fraction of failed trials.

To this end, we use model-based reinforcement learning (MBRL). In MBRL, a statistical model that approximates the world is maintained, and decisions are made assuming the model correctly approximates the world. The effects of the decisions are used to learn more about the world. By using a MBRL approach in our setting, from trial-and-error, we build and refine a statistical model that determines the likelihood of passing the property test for each candidate program.

Figure 7(a) depicts the iterations of our algorithm on Example 3.1. In contrast to the naive DD algorithm that invokes the property test 16 times (as already described in Figure 6), our algorithm only requires invoking it 10 times.

We present in detail each iteration of our algorithm using the standard decision tree model. In the first iteration, the current program is set to the original program P, and the initial dataset only contains a single example—that of P passing the property test. The initial model learned from the dataset predicts that any program can pass the test. Next, using the model, we prioritize candidates, which are the sub-programs that may be generated in the first iteration of DD—all the subsets of P whose size is 4 and their complements. The model arbitrarily chooses a subset $\langle$`f5`, `f6`, `f7`, `main`$\rangle$, and it fails the test. This result is added into the dataset, and the leftmost decision tree in Figure 7(b) is learned. Internal nodes represent conditions on the presence of specific functions, and leaf nodes correspond to probabilities of a given program's passing the property test. The first tree predicts that every program that contains `f4` will pass the property test, and that programs without `f4` will fail. In the next iteration, the subset $\langle$`f1`, `f2`, `f3`, `f4`$\rangle$, which is predicted to pass the test, is chosen and fails. This result is added into the dataset, and the next decision tree is learned. Now the number of partitions is doubled because all the subsets of size 4 (and their complements) failed the test. Now the model predicts that `main` as well as `f4` should be present in a desirable program. In the next iteration, the complement of $\langle$`f1`, `f2`$\rangle$, which is predicted to pass the test, is chosen and fails. This result is added into the dataset, and the next decision tree is learned. Now the model predicts that `main` as well as `f2` should be present in a desirable program. In this manner, after 6 iterations, the desirable decision tree is learned; it regards programs containing both `main` and `f1` as desirable.

In the 7th and 8th iterations, the depicted two programs are chosen, as they are the smallest among the available next candidates. The program comprising `main` and `f1` passes the property test. The results are added into the dataset. However, because the observations do not contradict the model, the model is not updated. The available next candidates are $\langle$`f1`$\rangle$ and $\langle$`main`$\rangle$, both of which fail the property test. By the last two iterations, it is confirmed that removing a single element from the current program $\langle$`f1`, `main`$\rangle$ does not pass the property test. Therefore, the loop terminates and the program $\langle$`f1`, `main`$\rangle$ is returned as the minimal one.

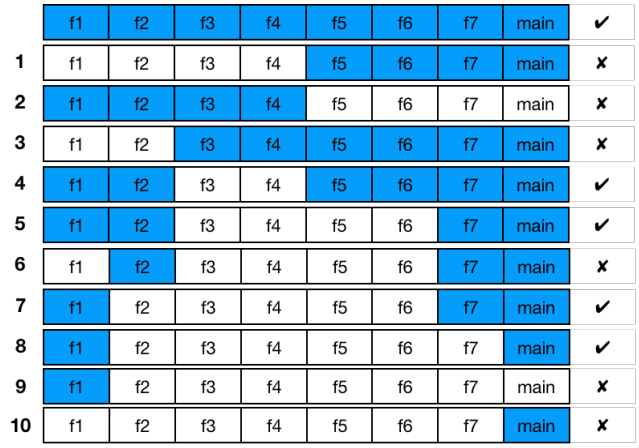
4.2 Markov Decision Process for Delta Debugging

We formalize a Markov decision process (MDP) for delta debugging. Given an original program P and a property test function O, the goal of this MDP is to find a good policy that guides the delta debugging algorithm towards a minimal program that satisfies O. Throughout this section, we will use the terms program and list interchangeably as DD views a program as a list of elements. Each component of this MDP is defined as follows:

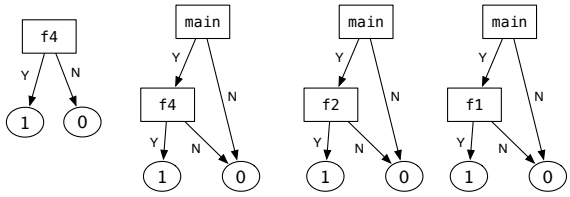
- *State*: The set of states $\mathcal{S}$ is a set of pairs of a current candidate program and a current level of granularity (i.e., the number of partitions) denoted $\mathbb{P} \times \mathbb{N}$ where $\mathbb{P}$ is the universe of all possible programs. The initial state s_0 is $\langle L, 2\rangle$ where L is a list representation of the original program P.
- *Action*: The set of actions $\mathcal{A}$ is the same as the set of states, and the set of possible actions $A(s)$ at a state $s = \langle L, n\rangle$ is defined as follows:

$$\begin{aligned} A(s) = {} & \{\langle\langle u_1\rangle, 2\rangle, \cdots, \langle\langle u_n\rangle, 2\rangle\} && \text{(subsets)} \\ & \cup \{\langle L \setminus u_1, n-1\rangle, \cdots, \langle L \setminus u_n, n-1\rangle\} && \text{(complements)} \\ & \cup \{\langle L, 2n\rangle\} && \text{(more granularity)} \end{aligned}$$

where $u_1, \cdots, u_n$ are n partitions of L. That is, the set $A(s)$ consists of all the possible pairs of a next candidate program and granularity in the current state s (i.e., all the right hand sides that may appear at lines 6, 8, and 10 during the iterations of Algorithm 1).



(a) Iterations of our algorithm.



(b) Decision trees learned after 1st, 2nd, 3rd, and 6th iterations, respectively.

Figure 7: Running Example. After 6 iterations, the desirable decision tree is learned. The minimal program is found in 10 iterations.

- *Transition function*: The transition function T is defined as follows (where $s = \langle L, n\rangle$, $\langle u_1, \cdots, u_n\rangle$ are n partitions of L, and $a = s'$):

$$T(s' \mid s, a) = \begin{cases} 1 & (s' = \langle u_i, 2\rangle, O(u_i) = \mathsf{T}) \\ 1 & (s' = \langle L \setminus u_i, n-1\rangle, O(L \setminus u_i) = \mathsf{T}) \\ 1 & (s' = \langle L, 2n\rangle, \nexists i.\ O(u_i) = \mathsf{T} \vee O(L \setminus u_i) = \mathsf{T}) \\ 0 & \text{(otherwise)} \end{cases}$$

  In other words, a state transition occurs only when either a next candidate program exhibits the desired property (the first two cases) or more granularity in dividing programs is necessary since none of them are desirable (the third case).
- *Reward function*: The reward function R is defined as follows:

$$R(\langle L, n\rangle) = \begin{cases} 1 & (L \text{ is 1-minimal}) \\ 0 & \text{(otherwise)} \end{cases}$$

  A reward is given at state $\langle L, n\rangle$ iff L is a 1-minimal program. Checking 1-minimality of L requires the test function O to guarantee that any variant derived from L by removing a single element does not pass the property test while L does.

Intuitively, the goal of the MDP described above is to find a 1-minimal solution with the smallest number of transitions. Unfortunately, solving the MDP (i.e., learning the optimal policy) is impractical. It requires a large number of invocations to the transition function T and reward function R defined using O that incurs nontrivial computation cost.

To address this issue, we learn a sub-optimal policy using the model-based reinforcement learning method [38]. We simultaneously learn a probabilistic model $\mathcal{M} : \mathbb{P} \to [0, 1]$ that returns a probability of a given candidate program's passing the property test function and derive a policy from the model. We will denote $\hat{T}$ and $\hat{R}$ as approximations of T and R, respectively, and they are defined using $\mathcal{M}$ instead of O. The function $\hat{T}$ is defined as follows (where $s = \langle L, n\rangle$, $\langle u_1, \cdots, u_n\rangle$ are n partitions of L, and $a = s'$):

$$\hat{T}(s' \mid s, a) = \begin{cases} \mathcal{M}(u_i) \,/\, K_{s,a} & (s' = \langle u_i, 2\rangle) \\ \mathcal{M}(L \setminus u_i) \,/\, K_{s,a} & (s' = \langle L \setminus u_i,\ n-1\rangle) \\ \left(\prod_{\langle L', n'\rangle \in A(s) \setminus s'} 1 - \mathcal{M}(L')\right) / K_{s,a} & (s' = \langle L, 2n\rangle,\ 2n \le |L|) \\ 0 & (s' = \langle L, 2n\rangle,\ 2n > |L|) \end{cases}$$

where $K_{s,a}$ is a normalization factor to make $\hat{T}$ a probability distribution. In the first two cases, the transition probability is defined as the probability of a target subset or its complement's passing the property test. The other two cases are for increasing granularity. Recall that we increase the granularity only when none of the next candidates pass the property test. We compute the probability of such a case as the probability of all the next candidate programs failing the property test under the model $\mathcal{M}$. However, we may be in a situation where the model misguides us to carelessly increase the granularity until the algorithm terminates and return a non-minimal program. To prevent such cases, the probability of increasing the granularity is 0 if the current granularity is the finest one, which is described in the last case of the above definition. By doing so, the algorithm will try all the subsets and complements before it terminates, guaranteeing 1-minimality.

The function $\hat{R}$ is defined as follows:

$$\hat{R}(\langle L, n\rangle) = \prod_{1 \le i \le |L|} \Big(1 - \mathcal{M}(L \setminus u_i)\Big)$$

where $\langle u_1, \cdots, u_{|L|}\rangle$ are $|L|$ partitions of L. In other words, $\hat{R}(\langle L, n\rangle)$ is the probability of L's being 1-minimal under the model $\mathcal{M}$.

Putting it all together, given approximated functions $\hat{T}$ and $\hat{R}$, our goal is to learn the following optimal policy:

$$\hat{\pi}(s) = \arg\max_{a \in A(s)} \sum_{s'} \hat{T}(s' \mid s, a)\hat{V}(s') \tag{1}$$

where $\hat{V}$ is the expected sum of rewards under the policy $\hat{\pi}$:

$$\hat{V}(s) = \hat{R}(s) + \gamma \sum_{s'} \hat{T}(s' \mid s, \hat{\pi}(s))\hat{V}(s') \quad (0 \le \gamma < 1). \tag{2}$$

$\hat{V}$ can be computed via dynamic programming. Based on the policy, an optimal action is chosen. State transitions and rewards caused from the action will be used to refine the appoximations, which will be used to also improve the policy again. In our evaluation, we determined that $\gamma = 0$ yields the best performance (i.e., the computation of $\hat{V}$ only evaluates the immediate rewards).

4.3 Statistical Models

We describe how to learn the aforementioned model. Our goal is to use the model to predict a probability of $O(P)$ for a given program P. We learn the model from data collected during the program debloating process.

Feature Representation. Suppose a program P is represented as a list of n elements:

$$\langle u_1, \ldots, u_n \rangle$$

Every sub-program P' of P is encoded as a binary feature vector of length n by a feature encoding function F:

$$F(P') = \langle b_1, \ldots, b_n \rangle$$

where b_i is 1 if u_i is included in the sub-program P', and 0 otherwise. For example, the original program P itself is encoded as 1^n and the empty program is encoded as 0^n.

Labeled Data. Each feature vector is labeled with a boolean value, which is the result of the property test. For example, the feature vector $F(P)$ of a program P is $O(P)$. Such labeled data are collected from each trial during the DD process.

Learning a Model. We learn a statistical model $\mathcal{M}$ using an off-the-shelf supervised learning algorithm such as the decision tree using the feature vectors and labels collected during the DD process. Such learned models predict a probability of $O(P)$ for a given feature vector representation of P. The approximated versions of transition ($\hat{T}$) and reward ($\hat{R}$) functions are defined by replacing O with $\mathcal{M}$ in their original definitions.

4.4 Learning-based Program Debloating

We now describe our overall algorithm presented in Algorithm 2. It is given an original program P to be reduced, a property test function O, an off-the-shelf supervised learning algorithm $\mathcal{L}$ for learning a model, and a feature encoding function F. The state s denotes a current state throughout the algorithm.

At line 2, the state is initialized as the original program. At line 3, a dataset is initialized to be a pair of a feature vector encoding the original program ($F(P) = 1^{|P|}$) and its label (T). During the iterations, the model is iteratively refined at line 5 using the updated dataset. The main loop (lines 4–13) iterates until the program cannot be split any further. At each iteration of the loop, the approximated policy is computed using the current model (line 6). Using the learned policy, a next state is chosen (line 7), and its property is checked (line 8). We change the current state only if the new program still exhibits the property or we need more granularity in dividing the input program (line 9). The result of the property test is added into the dataset (line 12). The termination condition (line 14) holds when all the sub-programs of the current program have been tested. This algorithm guarantees 1-minimality thanks to the design of $\hat{T}$ described in Section 4.2.

5 EVALUATION

We experimentally evaluate Chisel by addressing the following research questions:

- **Q1. Effectiveness**: How effectively does Chisel reduce a given program in terms of reduction quality and reduction time?
- **Q2. Security**: Can Chisel trim out known vulnerabilities in the programs? Can Chisel also reduce the potential attack surface of the programs?
- **Q3. Robustness**: How robust is the reduced program generated by Chisel against new unseen inputs?

Algorithm 2 Learning-guided Delta Debugging

Input: A program P
Input: A property test function O
Input: A function $\mathcal{L}$ for learning the probabilistic model
Input: A feature encoding function F
Output: A minimal program P' such that $O(P') = \mathsf{T}$

1: $L \leftarrow$ A list representation of P
2: $n \leftarrow 2$ ▷ Initial state
3: $\mathcal{D} \leftarrow \{\langle F(L), \mathsf{T}\rangle\}$ ▷ Initial dataset
4: **repeat**
5: $\quad \mathcal{M} \leftarrow \mathcal{L}(\mathcal{D})$ ▷ Learn a probabilistic model
6: $\quad$ Construct $\hat{\pi}$ using $\mathcal{M}$ and the equations (1) and (2)
7: $\quad \langle L', n' \rangle \leftarrow \hat{\pi}(\langle L, n \rangle)$ ▷ Next action from $\hat{\pi}$ defined in the eq. (1)
8: $\quad \diamond \leftarrow O(L')$ ▷ $\diamond \in \{\mathsf{T}, \mathsf{F}\}$
9: $\quad$ **if** $\diamond = \mathsf{T}$ or $n' = 2n$ **then**
10: $\quad\quad \langle L, n \rangle \leftarrow \langle L', n' \rangle$ ▷ State transition
11: $\quad$ **end if**
12: $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{\langle F(L'), \diamond \rangle\}$ ▷ Add the new data into the dataset
13: **until** $n \leq |L|$
14: **return** P' corresponding to L

5.1 Setting

Implementation. We instantiate Chisel as a program reducer for C programs based on the syntax-guided hierarchical delta debugging algorithm [37]. Chisel first reduces global-level components (i.e., global variable declarations, type definitions, function definitions, etc.) with the standard delta debugging algorithm, and recursively applies it local-level components (i.e., assignments, if-statements, while-statements, etc.). Once the local-level reduction finishes, Chisel re-invokes the global-level reduction and continues the whole process until a minimal version is found. Chisel simply rejects nonsensical programs without invoking the test script by using a simple dependency analysis, such as programs that do not contain the main function, variable declarations, variable initializations, or return statements.

Chisel consists of 2,361 lines of OCaml code. We used an off-the-shelf decision tree algorithm called FastDT[3] to learn models. We learn exact decision trees (i.e., we neither use boosting/bagging nor bound the maximum depth of a tree). All experiments were conducted on Linux machines with 3.0 GHz and 128 GB memory.

Benchmarks. We evaluate Chisel on a suite of 10 benchmark programs from GNU packages. Table 1 shows the characteristics of these programs. These programs were chosen because they are open-source, widely used programs, each of them contains known vulnerabilities reported as CVEs, and their manually reduced implementations are available in BusyBox [3], a lightweight UNIX utility package for embedded systems. All the numbers are measured after macro expansion.

Specifications. The desired features to preserve are chosen with reference to the BusyBox implementations of the benchmarks. We assume that the options supported by the default configuration of BusyBox are the core functionalities of each program. In addition, for security reasons, we forced the reduced programs not to result in any undefined behaviors (not just crashes) such as buffer overrun or uninitialized variable use even during the executions for non-core

[3] http://legacydirs.umiacs.umd.edu/~hal/FastDT

Table 1: Characteristics of the benchmark programs. LOC, #Func, and #Stmt reports the lines of code, the number of functions, and the number of statements after macro expansion.

Program	LOC	#Func	#Stmt	CVE ID (CVSS Score)	Vulnerability
bzip-1.05	18,688	108	6,532	CVE-2011-4089 (4.6)	Executing arbitrary code by pre-creating a temporary directory.
chown-8.2	69,894	757	24,792	CVE-2017-18018 (1.9)	Modifying the ownership of arbitrary files with the "-R -L" option.
date-8.21	75,898	878	26,147	CVE-2014-9471 (7.5)	Executing arbitrary code with the "-d" option.
grep-2.19	49,011	432	12,138	CVE-2015-1345 (2.1)	Causing a crash with the "-F" option.
gzip-1.2.4	13,223	93	4,118	CVE-2005-1228 (5.0)	Writing to arbitrary directories with the "-N" option.
mkdir-5.2.1	28,202	263	10,679	CVE-2005-1039 (3.7)	Modifying the ownership of arbitrary files with the "-m" option.
rm-8.4	89,694	764	27,695	CVE-2015-1865 (3.3)	Modifying the ownership of arbitrary files with the "-rf" option.
sort-8.16	71,315	753	24,890	CVE-2013-0221 (4.3)	Causing a crash with the "-d" or "-M" option.
tar-1.14	45,778	502	13,227	CVE-2016-6321 (5.0)	Writing to arbitrary files.
uniq-8.16	64,915	665	22,086	CVE-2013-0222 (2.1)	Causing a crash with a long input string.
Total	516,644	5,215	172,304		

functionalities. To this end, we compiled the programs with the sanitizer options [4] of Clang and monitored undefined behaviors at runtime. To extensively exercise all the functionalities, we collected test cases from the original source code repositories. In summary, Chisel generates a reduced version of the program that satisfies the following constraints:

- The reduced program must be compilable.
- The reduced program must have the same output as that of the original program for the core functionalities.
- The reduced program must not yield undefined behaviors for the non-core functionalities.

In addition, we set timeouts for each execution to 0.01–1 seconds depending on the running cost of each benchmark. This is to prevent running non-terminating programs that are generated when Chisel introduces infinite loops.

Baseline Reducers. We compare Chisel to two state-of-the-art program reduction approaches: C-Reduce [36] and Perses [37]. C-Reduce is an off-the-shelf C program reducer. We implemented Perses based on the recent work of Sun et al. [37]. Like Chisel, Perses also reduces programs with respect to the grammar, but their reduction process is not guided by a probabilistic model. All three tools are based on variants of delta-debugging and guarantee that the reduced program is 1-minimal.

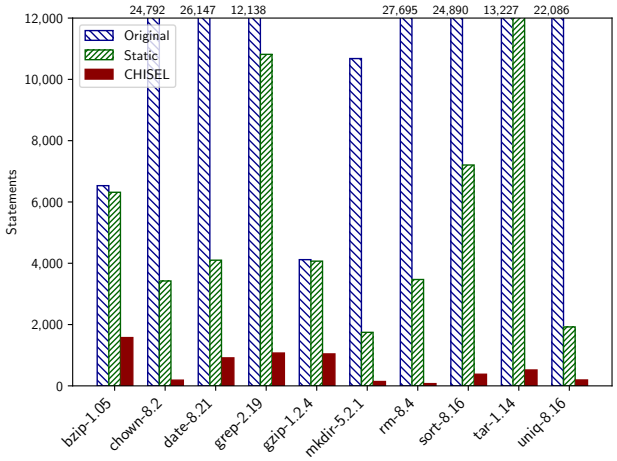


Figure 8: Effectiveness of Chisel in terms of reduction ratio. Original reports the number of statements in the original programs. Static and Chisel report those in the programs after unreachable function removal by static analysis and program debloating by Chisel, respectively.

5.2 Effectiveness of Reduction

We first evaluate the effectiveness of Chisel in terms of reduction size. We measured the number of statements of the original programs, the reduced versions by unreachable code removal, and the ones generated by Chisel. For unreachable code removal, we removed all the unreachable functions from the main function using Sparrow [13], a static analyzer for C programs. Figure 8 shows the results. First of all, the static analysis reduced the number of statements from 172,304 to only 55,848 (32.4%). The reason for the huge reduction even with static reachability is mainly due to a large amount of library code that all the GNU CoreUtil programs share. Among the statically reachable statements, Chisel further reduced 89.1% of statements and resulted in only 6,111.

Next, we evaluate the running time of Chisel compared to existing program reducers in Figure 9. Chisel effectively reduced all of the benchmark programs within a timeout limit of 12 hours. C-Reduce and Perses ran out of time for 6 programs and 2 programs respectively. These results show that our learning-based approach is more efficient than the previous ones. Perses, which is a purely grammar-based program reducer, is faster than C-Reduce, which is basically a line-based reducer, by avoiding a large number of trials with syntactic errors. Chisel outperforms both of these tools on all the benchmarks because it not only avoids syntactic errors, akin to Perses, but it also learns to avoid semantic errors. As a result, Chisel runs up to 7.1x and 3.7x faster on average than C-Reduce and Perses, respectively.

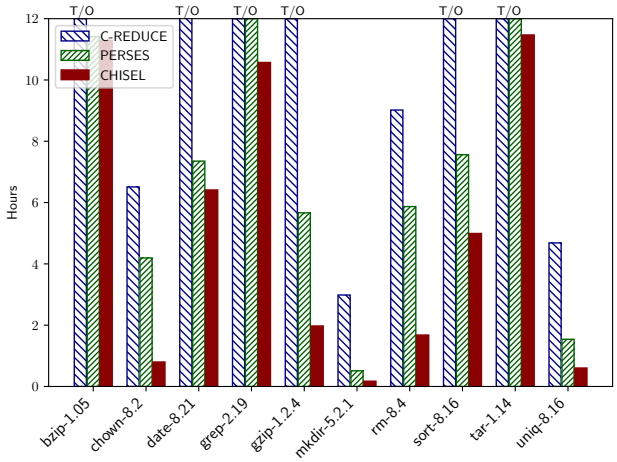


Figure 9: Effectiveness of Chisel in terms of running time. C-Reduce, Perses, and Chisel report the running time of program debloating by each tool. The timeout (T/O) is set to 12 hours.

Table 2: Comparison to BusyBox. #Opt reports the number of command-line options supported by the original GNU programs and their BusyBox counterparts. #Stmt reports the number of statements of each program.

	GNU CoreUtil		BusyBox		Chisel
Program	**#Opt**	**#Stmt**	**#Opt**	**#Stmt**	**#Stmt**
bzip-1.05	15	6,316	5	2,342	1,575
chown-8.2	15	3,422	2	141	186
date-8.21	11	4,100	7	107	913
grep-2.19	45	10,816	16	355	1,071
gzip-1.2.4	18	4,069	3	1,058	1,042
mkdir-5.2.1	7	1,746	2	94	142
rm-8.4	12	3,470	3	89	73
sort-8.16	31	7,206	7	89	379
tar-1.14	97	12,780	8	403	538
uniq-8.16	12	1,923	7	51	192
Total	263	55,848	60	4,729	6,111

We also justify the reduction of the benchmark programs by comparing to their manually reduced implementations in the BusyBox package [3]. BusyBox is a single executable program that invokes all the utilities via subcommands. In this experiment, we excluded the large boilerplate code for parsing the subcommands and only measured the size of each of the subprograms. Table 2 shows the comparison. In total, the original GNU CoreUtil programs implement 263 command-line options in 55,848 statements, while BusyBox has 60 options and 4,729 statements. The reduced programs by Chisel, which have the same options as BusyBox, comprise 6,111 statements. Of the 10 programs, the BusyBox implementations have a fewer number of statements for 7 programs since

```
1  void initseq(struct seq* seq) {
2      seq->count = 0;
3      seq->lines = 0;
4  }
5
6  void system_join(FILE* fp1, FILE* fp2) {
7      struct seq seq1, seq2;
8      initseq(&seq1);
9      getseq(fp1, &seq1);
10     initseq(&seq2);
11     getseq(fp2, &seq2);
12 }
13
14 int main() {
15     fp1 = fopen(name[0], "r");
16     fp2 = fopen(name[1], "r");
17     system_join(fp1, fp2);
18     return 0;
19 }
```

(a)

```
1  struct seq seq1, seq2; // All fields are automatically initialized to 0
2
3  int main() {
4      fp1 = fopen(name[0], "r");
5      fp2 = fopen(name[1], "r");
6      getseq(fp1, &seq1);
7      getseq(fp2, &seq2);
8      return 0;
9  }
```

(b)

Figure 10: Code snippets from the reduced versions of join from Chisel (a) and C-Reduce (b). Chisel generates a program that preserves locality and modularity while C-Reduce mangles common software engineering practices.

humans write more optimized code. However, the aspects sometimes vary depending on their low-level algorithms, data-structures, and programming styles. Overall, Chisel generates reasonable code in terms of size.

Chisel yields more natural programs compared to existing reducers such as C-Reduce, which considers programs as string input data and whose primary goal is to minimize (possibly randomly generated) crashing inputs (i.e., programs) to compilers. Therefore, C-Reduce does not heed to common software engineering practices such as modularity and locality. On the other hand, one of our goals is to generate natural programs that humans can maintain and extend. Figure 10 depicts an example. Notice that Chisel preserves the structures of the original source code but C-Reduce breaks modularity and locality.

The reader may wonder whether a naive approach to program reduction based on runtime code coverage suffices in practice. To justify the need to use Chisel, we also compared the runtime code coverage of the original and reduced programs. This coverage is computed using llvm-cov [11] with the default options.

Table 3: Dynamic code coverage. #Line and Coverage report the number of executable lines and covered lines by the default functionality.

	Original		Reduced	
Program	**#Lines**	**Coverage**	**#Lines**	**Coverage**
bzip-1.05	6,605	4,520 (68.4%)	1,586	1,568 (100.0%)
chown-8.2	3,895	1,580 (40.6%)	195	192 (98.5%)
date-8.21	4,403	2,007 (45.6%)	953	946 (99.3%)
grep-2.19	12,011	4,304 (35.8%)	1,118	961 (86.0%)
gzip-1.2.4	4,450	2,290 (51.5%)	1,079	1,054 (97.7%)
mkdir-5.2.1	1,841	562 (30.5%)	164	159 (97.0%)
rm-8.4	3,915	1,586 (40.5%)	75	75 (100.0%)
sort-8.16	7,836	3,451 (44.0%)	403	399 (99.0%)
tar-1.14	14,092	2,524 (17.9%)	527	510 (96.8%)
uniq-8.16	2,103	858 (40.8%)	205	204 (99.5%)
Total	61,151	23,681 (38.7%)	6,305	6,086 (96.5%)

Table 4: Security hardening. CVE shows whether the known CVEs are disabled by Chisel. #Gadgets and #Alarms report the number of ROP gadgets and static analysis alarms of the original and reduced programs.

		#Gadgets		#Alarms	
Program	**CVE**	**Original**	**Reduced**	**Original**	**Reduced**
bzip-1.05	✗	662	298 (55.0%)	1,991	33 (98.3%)
chown-8.2	✓	534	162 (69.7%)	47	1 (97.9%)
date-8.21	✓	479	233 (51.4%)	201	23 (88.6%)
grep-2.19	✓	1,065	411 (61.4%)	619	31 (95.0%)
gzip-1.2.4	✓	456	340 (25.4%)	326	128 (60.7%)
mkdir-5.2.1	✗	229	124 (45.9%)	43	2 (95.3%)
rm-8.4	✗	565	95 (83.2%)	48	0 (100.0%)
sort-8.16	✓	885	210 (76.3%)	673	5 (99.3%)
tar-1.14	✓	1,528	303 (80.2%)	1,290	19 (98.5%)
uniq-8.16	✗	349	109 (68.8%)	60	1 (98.3%)
Total		6,752	2,285 (66.2%)	5,298	243 (95.4%)

Table 3 reports the number of executed lines for each benchmark. In total, the execution of the original programs with the core functionalities covered 23,681 (38.7%) lines among 61,151 executable lines. That is much larger than the lines of executable code in the reduced programs. Among 6,305 executable lines, the execution of the reduced programs covered 6,086 (96.5%) lines. In summary, these results show that Chisel can reduce significantly more code than a naive approach based on dynamic code coverage.

5.3 Security Hardening

We next report upon the efficacy of Chisel in terms of security hardening. We evaluate this aspect using three different means. First, we inspected the reduced programs and checked whether the known vulnerabilities are trimmed by Chisel. Second, we measured the reduction in the potential attack space by counting the number of gadgets in the original and reduced programs using ROPGadget [12]. Third, we ran a state-of-the-art static buffer overrun analyzer, Sparrow [13], and inspected all reported alarms.

The results are shown in Table 4. Chisel successfully eliminated the CVEs in 6 of the 10 programs. Of these, 4 CVEs were in non-core functionalities, and more significantly, 2 CVEs were in core functionalities (tar-1.14 and date-8.21). CVEs in core functionalities can be typically removed when the vulnerabilities are triggered in corner cases. As we saw in the case of tar-1.14 in Section 2, extracting tar balls of files whose names contain '../' is not common and rarely exercised by test cases. In that case, Chisel can aggressively remove such sub-functionalities handling corner cases even in core functionalities (i.e., extracting tarball). Chisel could not eliminate the CVEs in the remaining 4 programs because the vulnerabilities are triggered with the core functionalities and are not easily fixable by reduction (e.g., race condition). In addition, Chisel reduced potential attack surfaces by removing 66.2% of ROP gadgets on average. The decreased size and complexity of the reduced programs also enable to apply more precise program checkers such as static analyzers. The debloating reduced the number of buffer overrun alarms reported by Sparrow by 95.4%, making it feasible to manually inspect the few remaining alarms, all of which turned out to be false upon inspection.

5.4 Robustness

We measured the robustness of the debloated programs by running a state-of-the-art fuzzer, AFL [1]. The programs were tested with randomly generated command-line inputs and input files (if necessary) by the fuzzer for three days.

In most cases, the debloated programs are robust since no crashes were detected by fuzzing, despite the fact that the test cases written by the original developers extensively exercise the whole programs. In addition, the runtime monitoring using the sanitizers effectively filtered out erroneous programs during the debloating process. In our experience, debloating without the sanitizers yields problematic programs. Common bugs in C programs such as uninitialized variable or buffer-overflow cause the program to crash non-deterministically, and thus the test script returns non-deterministic results. However, the sanitizers catch such erroneous behaviors during execution, enabling Chisel to avoid candidate programs that contain such bugs.

Nevertheless, the reduced programs may fail with a new unseen input. In our experience, random fuzzing can help enrich the test script and derive robust programs. For grep-2.19 and bzip-1.0.5, the fuzzer quickly found crashing inputs within 10 minutes. Figure 11 shows a typical example excerpted from the code of grep-2.19. Initially, the shaded part was removed without any violation, since all the test cases were small so that re-allocating a new memory chunk was not needed. This initial program can be easily crashed (i.e., by a buffer-overrun) by random inputs generated by the fuzzer. In that case, we simply added the inputs into the test script; re-ran Chisel; and re-ran the fuzzer. This simple feedback process effectively improved the robustness of the resulting program. Unlike program synthesis [22] that often overfits the inputs, program reduction can easily find general enough programs if the original

```
1  struct dfa {
2      token* tokens; /* Tokens */
3      int talloc; /* Token buffer size */
4      int tindex; /* Token index */
5      /* Omitted for clarity */
6  };
7
8  struct dfa *dfa;
9
10 void add_tok (token t) {
11     if (dfa->talloc == dfa->tindex)
12         dfa->tokens = (token *) realloc (/* larger size */);
13     *(dfa->tokens + (dfa->tindex++)) = t;
14 }
```

Figure 11: Code snippet of grep-2.19. The shaded code was removed in the first trial.

program is correct. As a result, the fuzzer did not find any crash input for the next version for three days.

Summary of Results. In summary, Chisel efficiently reduced large C programs, yielding up to 7.1x and 3.7x speedup compared to C-Reduce and Perses, respectively. C-Reduce and Perses ran out of time for 6 programs and 2 programs respectively among the 10 benchmarks. Chisel removed 88.1% of code that is even deemed reachable by a sophisticated static analysis. It could also eliminate known CVEs and ROP gadgets. The robustness of the reduced programs was empirically confirmed by running a static analyzer and a random fuzzer.

6 THREATS TO VALIDITY

There are several threats to the validity of our approach. We outline these next along with proposals to mitigate them.

- **Non-determinism in test scripts:** Chisel may misbehave if the test script returns non-deterministic results. Such behaviors are mainly caused by undefined behaviors (e.g., uninitialized variables) of candidate programs. As already described in Section 5.4, we have partially solved this problem by using the sanitizers to prevent undefined behaviors. However, the sanitizers may miss erroneous behaviors if a target program uses external libraries that are not compiled with the same protection schemes. This problem can be mitigated by recompiling external libraries with the same schemes, if the library sources are available. The other reason for non-deterministic behaviors is caused by our method to avoid non-terminating candidate programs. Some transformations may lead to non-terminations (e.g., removing loop termination conditions) and to reject such invalid programs, we set a timeout limit. However, if such a limit is not large enough, the test script may return non-deterministic results even for a program of which execution time has a slight variance. This issue can be mitigated by dynamically detecting and escaping infinite loops [20] without setting timeout limits.
- **Incompleteness of test inputs:** We tested the robustness of the resulting programs using the AFL fuzzer. However, our testing may not be exhaustive enough if inputs are required to be in a specific format, since we cannot provide a specific grammar to AFL that could enable it to prune the search space by filtering out ill-formed test inputs. We can mitigate this issue by using grammar-based fuzzers [25, 44].
- **Unsoundness of static analysis:** We used the Sparrow static analyzer to test robustness along with AFL. Although the analyzer is sound with respect to most features of C programs, it may be unsound if a target program exhibits tricky features such as complex pointer arithmetic operations or complex control flows caused by API functions of unknown semantics. In practice, since designing a fully sound static analyzer is extremely challenging, various static analyzers that are *soundy* [31] with respect to different language features are used. We can mitigate the issue by combining the results of multiple static analyzers that possess different capabilities in this regard.

7 RELATED WORK

Program Debloating. Recently, a large body of work has proposed techniques to alleviate software bloat at different levels of granularity. For coarse-grained (i.e., application-level) debloating, Rastogi et al. [35] propose a technique to debloating application containers running on Docker [6]. They decompose a complicated container into multiple simpler containers with respect to a given user-defined constraint. Their technique is based on dynamic analysis to obtain information about application behaviors.

Techniques have also been proposed for finer-grained debloating. JRed [28] and RedDroid [27] trim unused methods and classes from Java and Android applications, respectively. Quach et al. [34] present a debloating system that trims out unnecessary code during compilation and loading. Their system reduces applications and libraries based on function-level dependencies that are computed using static analyses and training-based techniques. For each program, all shared libraries and invoked functions are learned. Then, at load time, the loader loads only these libraries and functions.

Compared to the previous approaches, Chisel is applicable to program debloating at an even finer granularity such as statement-level. Existing fine-grained approaches based on static analysis [28, 34] are conservative in that they remove only unreachable code. Instead, our system aggressively removes redundant code even on the execution paths.

There is also research on software bloat detection. Bhattacharya et al. [19] introduce an analysis technique to detect sources of bloat in Java applications. It aims to detect statements that are possible sources of bloat when optional features are no longer required. It starts by assuming that methods are potential features from which an interaction graph is constructed. Next, the graph is traversed by some heuristic rules to find likely sources of bloat. Finally, a user must intervene for confirming the detected statements and removing them. Unlike their work, our approach automatically removes redundant code with respect to a given test script.

In addition to the work on software bloat detection, there is a large body of research on detecting and reducing runtime memory bloat [9, 33, 40–43]. These works have an orthogonal yet complementary goal to ours, since program debloating has the potential to mitigate this problem to a certain extent through removing code from execution paths.

Test-input Minimization. A large body of program reduction techniques have been proposed in the context of test-input minimization [23, 32, 36, 37]. Their goal is to minimize input programs that cause compilers or interpreters to crash. Since they do not intend the minimized source code to be executed and maintained, however, the resulting programs are not carefully generated in terms of either security or readability. For example, as already described in Section 5, C-Reduce [36] often mangles common software engineering practices. Our goal, on the other hand, is to reduce programs for subsequent use by developers.

Existing techniques blindly search towards the minimal programs with only simple guidance from an oracle. Recently, researchers have proposed efficient techniques that are aware of syntactic structures of programs [23, 32, 37]. While such prior knowledge allows avoiding a number of trials involving syntactically ill-formed programs, these approaches produce a large number of semantic errors, and they do not infer new knowledge from previous trials. In addition to avoiding syntactically invalid programs by employing such an approach [37], our system avoids semantically invalid programs by building a statistical model online based on feedback from the oracle to guide the search. Our learning approach is applicable to accelerate all the existing approaches including general delta debugging [45] with unstructured inputs.

Program Slicing and Reachability Analysis. Program slicing is a well-known technique for program reduction designed for specific purposes such as debugging, testing, compiler optimization, software customization, or program analysis [17, 18, 21, 26, 29, 30, 39]. A slice is a sub-part of a program that is relevant to the value at some point of interest and is typically computed by static or dynamic dependencies. Our program debloating system aims at more general properties of interest and does not require specifying semantics and dependence relations. Moreover, our approach can be used to obtain smaller programs than those obtainable by static and dynamic slicing approaches.

In our experiments, we compared our approach with reachability analysis rather than slicing for the following reason. A dynamic slice contains all statements that affect the value of a target variable at a program point for a particular execution of the program. However, it may be challenging to determine variables and program points of interest from a high-level specification. Even if the user were to manually annotate such targets, dynamic slicing may still be ineffective for program debloating. For example, in Figure 4, the output of "`safer_name_suffix`" actually depends on most of the statements in the function. Therefore, it cannot remove the vulnerability, in contrast to Chisel.

Static reachability computation often results in imprecise approximations of actually reachable code because of its inherent limitations due to the undecidability of static analyses. It often cannot effectively handle complex control flows such as indirect procedure calls (e.g., setjmp / longjmp, function pointers, or reflection), complex conditionals, and pointer arithmetic. Our approach is not limited by such problems. Dynamic reachability computation can be more effective than the static approach in terms of code size. However, our study demonstrates that our approach results in programs even smaller than those based on dynamic reachability (Section 5) with smaller attack surfaces.

8 CONCLUSION

We presented Chisel, a learning-based system for program debloating based on delta debugging. Our approach effectively removed redundant code parts with respect to a given test script. The learned probabilistic model accelerated the process to find the minimal program. This debloating removed all vulnerabilities in the redundant functionalities and significantly reduced potential attack surfaces. Moreover, the reduced size and complexity enabled to apply precise program reasoning techniques and manual inspection.

In the future, we plan to extend Chisel in several directions, including investigating more expressive probabilistic models with efficient incremental learning, designing various forms of specification other than input-output examples, and applying to debloat programs written in arbitrary languages such as binary.

ACKNOWLEDGMENTS

We thank Aravind Machiry for providing insightful suggestions to evaluate the security and robustness of debloated programs. We thank the anonymous reviewers and Brian Heath for providing useful feedback to improve the paper. This research was supported by ONR award #N00014-18-1-2021, DARPA award #FA8750-15-2-0009, and NSF awards #1253867 and #1526270.

REFERENCES

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.
[2] App Distribution Guide - App Thinning (iOS, tvOS, watchOS). https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/AppThinning/AppThinning.html.
[3] BusyBox. https://busybox.net.
[4] Clagn 7 Documentation. https://clang.llvm.org/docs/.
[5] CVE-2016-6321. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6321.
[6] Docker. https://www.docker.com.
[7] EGLIBC. http://www.eglibc.org.
[8] GNU Tar extract pathname bypass. http://seclists.org/fulldisclosure/2016/Oct/96.
[9] Introducing Android Oreo (Go edition). https://www.android.com/versions/oreo-8-0/go-edition.
[10] Lighttpd. https://www.lighttpd.net.
[11] LLVM Commandline Guide. https://llvm.org/docs/CommandGuide/llvm-cov.html.
[12] ROPGadget. http://shell-storm.org/project/ROPgadget.
[13] Sparrow. https://github.com/ropas/sparrow.
[14] SQLite. https://www.sqlite.org.
[15] Toybox. http://landley.net/toybox.
[16] uClibc-ng. https://uclibc-ng.org.
[17] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*.
[18] Samuel Bates and Susan Horwitz. 1993. Incremental Program Testing Using Program Dependence Graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*.
[19] 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 & Applications (OOPSLA '13)*.
[20] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*.
[21] 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 '17)*.
[22] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* (2017).
[23] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-structured Test Inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*.
[24] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. 2016. On the Naturalness of Software. *Communications of ACM (CACM)*

(2016).

[25] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium (USENIX Security '12')*.

[26] Ranjit Jhala and Rupak Majumdar. 2005. Path Slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*.

[27] 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 IEEE International Symposium on Software Reliability Engineering (ISSRE '18)*.

[28] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the 40th IEEE Computer Society International Conference Computer on Software and Applications Conference (COMPSAC '16)*.

[29] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2015. A Preliminary Analysis and Case Study of Feature-Based Software Customization (Extended Abstract). In *IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*.

[30] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *17th IEEE International Symposium on High Assurance Systems Engineering (HASE '16)*.

[31] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM (CACM)* (2015).

[32] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*.

[33] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 2018. Understanding and Combating Memory Bloat in Managed Data-Intensive Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2018).

[34] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. *CoRR* abs/1802.00759 (2018). http://arxiv.org/abs/1802.00759

[35] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*.

[36] 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 '12)*.

[37] 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 '18)*.

[38] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*.

[39] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*.

[40] 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 '09)*.

[41] 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).

[42] 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*.

[43] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*.

[44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[45] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering (TSE)* (2002).