Fine Grained Dataflow Tracking with Proximal Gradients

Gabriel Ryan†, Abhishek Shah†, Dongdong She†, Koustubha Bhat‡, Suman Jana†
† Columbia University, ‡ Vrije Universiteit Amsterdam

Abstract

Dataflow tracking with Dynamic Taint Analysis (DTA) is an important method in systems security with many applications, including exploit analysis, guided fuzzing, and side-channel information leak detection. However, DTA is fundamentally limited by the Boolean nature of taint labels, which provide no information about the significance of detected dataflows and lead to false positives/negatives on complex real world programs.

We introduce proximal gradient analysis (PGA), a novel, theoretically grounded approach that can track more accurate and fine-grained dataflow information. PGA uses proximal gradients, a generalization of gradients for non-differentiable functions, to precisely compose gradients over non-differentiable operations in programs. Composing gradients over programs eliminates many of the dataflow propagation errors that occur in DTA and provides richer information about how each measured dataflow affects a program.

We compare our prototype PGA implementation to three state of the art DTA implementations on 7 real-world programs. Our results show that PGA can improve the F1 accuracy of data flow tracking by up to 33% over taint tracking (20% on average) without introducing any significant overhead (< 5% on average). We further demonstrate the effectiveness of PGA by discovering 22 bugs (20 confirmed by developers) and 2 side-channel leaks, and identifying exploitable dataflows in 19 existing CVEs in the tested programs.

1 Introduction

Dataflow analysis with dynamic taint analysis (DTA) is a fundamental building block in many common systems security tasks, such as automated vulnerability analysis, guided fuzzing, discovering information leaks, and malware analysis [4,14,20,38,42,56]. DTA analyzes dataflow between a specified set of sources and sinks in a program by instrumenting the program and tracking taint as it executes [30,34].

However, DTA is fundamentally limited by the Boolean information contained in taint labels: data either is tainted by a given source or not; there are no intermediate states or other sources of information. This means there is no way to identify and prioritize which dataflows are most significant. For example, given a series of operations

\[ x_1 = a \times 8; \ x_2 = b / 8; \ y = x_1 + x_2; \]

changes to the value of \( a \) will have a larger effect on the value of \( y \) than changes to value of \( b \), but taint labels cannot make this distinction. Moreover, it limits the ability of DTA frameworks to account for dataflows that are dependent on how operations compose. For example, in

\[ x_1 = x \times 2; \ x_2 = x_1 \& 1; \]

variable \( x_2 \) will only be affected by changes in the first bit of \( x_1 \), but changes to \( x \) will not affect \( x_2 \) due to the intermediate multiplication by 2.

While most DTA systems incorporate some special rules to handle these types of cases, we find in our evaluation (Section 5.2.1) that current DTA systems with these rules still make many errors in predicting dataflows, even at high compiler optimization levels that eliminate most intermediate operations. These errors have prevented DTA from being successfully applied in applications such as detecting keyloggers and memory corruption attacks [6,12,48,49].

The limitations of DTA led several researchers to propose Quantitative Information Flow (QIF) based methods as a more fine grained form of dataflow [33]. However, while QIF is able to track data more precisely, computing these measures is computationally expensive and does not scale effectively to large programs [26].

In this paper, we propose an alternate measure of dataflow that addresses the limitations of DTA while retaining its advantages in scalability. We observe that gradient, a multi-variate generalization of derivatives from elementary calculus, is a popular method for tracking the influence of inputs through differentiable models [16]. In particular, gradients have been used in neural
// taint source: x1, x2, x3
// taint sink: y

// input: x1, x2, x3 = 10
int y = x1 + x2*x2 - 100*x3;
if (y > THRESH) {
  // vulnerability
}

Figure 1: Example program in which gradient can guide a search to reach a vulnerability. While taint tracking identifies y as tainted by all three inputs, gradient measures the magnitude and direction of each influence, identifying that x3 is the most influential input and that minimizing it will maximize y due to its negative gradient.
2 Background

Our approach to gradient-based dataflow analysis draws on several techniques from the mathematical analysis and optimization literature. We provide a summary of the relevant methods below. We first summarize standard methods for computing gradients over compositions of smooth functions, and then review techniques from the non-smooth analysis literature that can be applied to computing gradients over programs.

2.1 Smooth Analysis

Gradients. The derivative for a smooth scalar function \( f(x) \) is defined as \( f'(x) = \lim_{\delta x \to 0} \frac{f(x+\delta x)-f(x)}{\delta x} \), where \( f: \mathbb{R} \to \mathbb{R} \). If a function has a derivative for all points in its domain, then it is considered a differentiable function. The gradient is a generalization of the derivative to multivariate functions, where \( f: \mathbb{R}^n \to \mathbb{R} \), \( f \) is defined as \( f = (f_1, \ldots, f_m) \), and \( \nabla f \) is the gradient of each of the \( m \) outputs: \( \nabla f: \mathbb{R}^n \to \mathbb{R}^{n \times m} \). For the rest of the paper, functions are multi-variate unless otherwise noted.

Chain Rule. Gradients of compositions of differentiable functions can be computed from gradients of the individual functions. This is known as the chain rule of calculus and is defined as follows, where \( \circ \) indicates the composition of two functions \( f \) and \( g \), and \( f' \) and \( g' \) are their respective gradients:

\[
(f \circ g)' = (f' \circ g) \circ g' \tag{1}
\]

Elementwise multiplication is used when \( f \) and \( g \) are multivariate.

Automatic Differentiation. Automatic Differentiation (AutoDiff) uses the chain rule to compute the gradient for potentially large compositions of differentiable functions. AutoDiff has been a longstanding tool in computational modeling and is a core component of deep learning frameworks such as Tensorflow [2,53]. However, existing AutoDiff methods and frameworks are limited to working with mostly continuous functions with limited discontinuity (e.g. ReLUs in neural networks).

2.2 Non-smooth Analysis

Extensive work has been done in the field of mathematical analysis on methods for approximating gradients over non-smooth functions. In this section we consider general multivariate functions of type \( f: \mathbb{R}^n \to \mathbb{R} \). We first describe a generalized type of continuity, called Lipschitz continuity, that applies to non-smooth operations in programs, and then define a generalization of gradients that apply to Lipschitz continuous functions.

Lipschitz Continuity. A function is Lipschitz continuous if its output does not change too much for small changes in the input. Formally, a function \( f \) is Lipschitz continuous if there exists a constant \( K \) (called the Lipschitz constant) that bounds how much the value of \( f \) can change between any two points in its domain. Figure 3a shows a simple Lipschitz continuous function along with the corresponding Lipschitz constant. In general the operations in any useful computation will yield a Lipschitz continuous function.

Generalized Gradients. On Lipschitz continuous functions, generalized gradients are used to approximate gradients [13,40]. Generalized gradients consist of generalized directional derivatives, which evaluate the gradient in a single direction as shown in Figure 3b. A generalized directional derivative in a direction \( v \in \mathbb{R}^n \) is defined as follows:

\[
f'(x;v) = \lim_{\lambda \to 0} \sup_{y \to x, \lambda \neq 0} \frac{f(y+\lambda v) - f(y)}{\lambda} \tag{2}
\]

Here \( x \) and \( y \) are two points in the domain of \( f \) where \( x \) is the point the derivative is evaluated, and \( \lambda \) is a distance along the vector \( v \) that the derivative is taken in. The chain rule for directional derivatives with functions \( g: \mathbb{R}^n \to \mathbb{R}^n \) and \( f: \mathbb{R}^n \to \mathbb{R} \) is defined:

\[
(f \circ g)'(x;v) = f'(g(x);g'(x;v)) \tag{3}
\]

When applied to generalized directional derivatives the composing functions must be monotonic. Several relaxed versions of the chain rule apply to generalized derivatives under different weaker assumptions about the composite functions [22,31,52].

A generalized gradient is approximated with a set of directional derivatives based on a matrix \( V \in \mathbb{R}^{n \times p} =
At a high level, our gradient propagation framework, PGA, is similar to Autodiff, computing the gradient of each operation and using the results as inputs to the next gradient computation. However, unlike Autodiff, we approximate the gradients of discrete functions with proximal gradients.

**Proximal Gradients.** Since programs are generally composed of discrete operations on integers, we define a gradient approximation called proximal gradients that can be evaluated on these discrete functions. Proximal gradients use the minima of a function within a nearby region defined with a special operator called the proximal operator [36]. This can be evaluated on both discrete and continuous functions $f: X^n \to X$, where $X$ is a set with euclidean norm that can represent integers or floats.

$$\text{prox}_f(x) = \arg\min_y (f(y) + \frac{1}{2}\|x - y\|_2^2)$$

(6)

The notation $\arg\min_y$ indicates that the operator evaluates to the value of $y$ that minimizes the sum of the function $f(y)$ and the distance cost.

We use the proximal operator to compute each generalized directional derivative $f'_i(x; v)$. Given a function $f$ representing a program operation, we constrain the proximal operator from Eq. 6 to a direction $v$:

$$\text{prox}_f(x; v) = \arg\min_y (f(y) + \frac{1}{2}\|x - y\|_2^2)$$

where $y = x + tv : t \in \mathbb{N}, y \in X^n$

(7)

We then define the proximal directional derivative based on the difference with $\text{prox}_f(x; v)$ constrained in the direction $v$ and scaled by the direction magnitude $\|v\|_2$:

$$\text{prox}_f(x; v) = \frac{f(\text{prox}_f(x; v)) - f(x)}{|\text{prox}_f(x; v) - x|} \ast \|v\|_2$$

(8)

This takes the same form as the generalized directional derivative (Eq. 2), but evaluated with the proximal operator. A proximal gradient is defined for a set of direction vectors $V$ like the generalized gradient (Eq. 4) using proximal directional derivatives:

$$\text{prox}_f(x; V) = [\text{prox}_f(x; v_1), \ldots, \text{prox}_f(x; v_p)]$$

(9)

Using proximal gradients allows us to evaluate gradients on discrete operations in programs as if they were continuous nonsmooth functions and apply the associated chain rule for generalized gradients in Eq. 5. For the rest of this paper, we refer to ‘proximal gradients’ simply as ‘gradients’ unless otherwise specified.

### 3.1 Program Gradient Evaluation

To compute gradients over programs with PGA, we model a program as a discrete function $P: X^n \to X^n$, and model the program state $x \in X^n$ as a vector (e.g., $x$ could model a byte array of size $n$ representing the program memory and registers). $P$ is composed of $N$ functions $P_i: X^n \to X^n, i \in \{1..N\}$ representing individual operations on the program state:

$$P(x) = P_N \circ P_{N-1} \circ \cdots \circ P_2 \circ P_1(x)$$

Each program operation $P_i$ is modeled as a combination of $n$ non-smooth scalar valued functions $f_{ij}: X^n \to X, i \in \{1..n\}$ that define how $P_i$ modifies each variable in the program state.

$$P_i(x) = [f_{i,1}(x), f_{i,2}(x), \ldots, f_{i,n}(x)]$$

We evaluate each $f_{ij}$ in $P'_i$ using the proximal directional derivative (Eq. 8):

$$P'_i(x; v) = [\text{prox}_{f_{i,1}}', (x; v), \ldots, \text{prox}_{f_{i,n}}', (x; v)]$$

To compose derivatives for a given operation $P_i$ from the previous operation $P_{i-1}$, we individually compose the derivatives of each $f_{ij}$ in $P_i$ from the previous operation $P_{i-1}$:

$$P_i \circ P_{i-1}'(x; v) = [\{[f_{i,1} \circ P_{i-1}'(x; v)], \ldots, [f_{i,n} \circ P_{i-1}'(x; v)]\}]$$

(10)

where each $f_{ij} \circ P_{i-1}'(x; v)$ is defined based on the directional derivative chain rule in Eq. 5:

$$(f_{ij} \circ P_{i-1}'(x; v) = f'_{ij} \circ (P_{i-1}(x); P_{i-1}'(x; v))$$
We then compute the proximal gradient using Eq. 9 for $x$%4 is determined by the derivative $\frac{dx}{d\text{input}}$. Samples are first collected at intervals of 2 and then used to compute the max/min directional derivative.

Using the chain rule from Eq. 10, we can compute a directional derivative for each final state of the program $P$ by chaining derivatives of the individual operations.

$$P'(x; v) = (P_N \circ P_{N-1} \circ \cdots \circ P_2 \circ P_1)'(x; v)$$

We then compute the proximal gradient using Eq. 9 for each program state by combining derivatives for a set of direction vectors represented by a matrix $V$:

$$P'(x; V) = [P'(x; v_1), P'(x; v_2), \ldots, P'(x; v_p)]$$

This is the same approach used in Automatic Differentiation, but extended to discrete functions and generalized gradients. This chained gradient approximation is designed to be error-free for all locally Lipschitz convex functions as well as some locally Lipschitz non-convex functions that meet the requirements for the non-smooth chain rule (e.g., monotonicity).

### 3.2 Proximal Derivative Evaluation

When applying proximal directional derivatives in practice, we make two modifications to the proximal directional derivative defined in Eq. 8 to model program behavior more closely.

First, we only consider the inputs to the operation itself in a function $f^* : X^k \rightarrow X, k \in \{1..n\}$ and associated $v^*$, where $k$ is the number of inputs to the operation. To simplify notation we drop the $*$, and for the rest of the paper assume $f$ and $v$ to refer to their $k$ dimensional variants on the current operation.

Second, we modify the proximal operator to select a nearby point that maximizes absolute change in $f$, which we denote $|f|$:

$$\text{prox}_{|f|}(x; v) = \arg\min_y (-|f(x) - f(y)| + \frac{1}{2}\|x - y\|_2^2)$$

This modified proximal operator selects the largest generalized derivative of $f$ based on either the maximum or minimum of $f$ in the direction $v$ (these correspond to the supremum or infimum of a generalized derivative). Accounting for both is necessary in dataflow analysis to avoid missing possible dataflows.

#### Proximal Derivative Algorithm

Algorithm 1 defines how we compute the proximal derivative for an operation $op$ that has two input variables $x_1$ and $x_2$, and returns an output $y$. We denote the derivatives of the inputs $x_1$ and $x_2$ and output $y$ to be $dx_1$, $dx_2$, and $dy$, where $dx_1$ and $dx_2$ are components of $v$, and $dy$ is computed using the proximal derivative with a maximum
We define a general framework for propagating derivatives over 5 abstract classes of operations that need to be handled in program analysis: floating point operations, integer valued operations, loading and storing variables, branching, and function calls to external libraries.

1. **Floating point operations**: We treat floating point operations as continuous functions and apply the standard chain rule (Eq. 10) with their analytic derivatives. If there are any potentially non-smooth floating point operations, such as floating point modulo, or typecasting between floating point types, we use proximal derivatives.

2. **Integer operations**: We consider any boolean or typecasting involving integers to be integer operations, as well as any arithmetic, bit shifting, or modulo on integer or pointer types. In general we use proximal derivatives on all integer operations, although in some cases such as arithmetic addition and multiplication we use analytic derivatives as an optional optimization.

3. **Load and Store**: When variables are stored or loaded from memory, their associated derivatives are also stored or loaded (our implementation uses shadow memory to track derivatives in memory, although any associative tracking mechanism could be used). If the memory address passed to a load instruction has a nonzero derivative, we set the derivative of the loaded variable to 1.0 if it does not already have a nonzero derivative. This is a simplifying approximation that may lead errors in evaluating the proximal gradient. However, we note that proximal derivatives on load operations can potentially be evaluated by sampling adjacent memory locations. We leave this to future work.

4. **Branches**: When dynamically computing derivatives, we can only reason about the derivative on the current execution path. If computing a derivative would require sampling an alternate execution path, we instead set that derivative to 0. Therefore, when a branch is encountered, we set any derivatives to 0 that are based on samples that would change the branch condition. This approach may miss some parts of the gradient but ensures we do not propagate incorrect derivatives. We note that sampling across multiple execution paths when handling branches could yield more accurate proximal derivatives and reason about control flow data flows (i.e. implicit data flows), we leave this to future work.

5. **External Library Functions**: Provided they do not have side effects, derivatives on external library function calls can be computed using proximal derivatives, while functions with side effects must be handled on a case by case basis. When an external function overwrites a buffer, we also clear the stored derivatives associated with that buffer.

### 3.3 Derivative Propagation Rules

We define a general framework for propagating derivatives over 5 abstract classes of operations that need to be handled in program analysis: floating point operations, integer valued operations, loading and storing variables, branching, and function calls to external libraries.

- **1. Floating point operations**: We treat floating point operations as continuous functions and apply the standard chain rule (Eq. 10) with their analytic derivatives. If there are any potentially non-smooth floating point operations, such as floating point modulo, or typecasting between floating point types, we use proximal derivatives.

- **2. Integer operations**: We consider any boolean or typecasting involving integers to be integer operations, as well as any arithmetic, bit shifting, or modulo on integer or pointer types. In general we use proximal derivatives on all integer operations, although in some cases such as arithmetic addition and multiplication we use analytic derivatives as an optional optimization.

- **3. Load and Store**: When variables are stored or loaded from memory, their associated derivatives are also stored or loaded (our implementation uses shadow memory to track derivatives in memory, although any associative tracking mechanism could be used). If the memory address passed to a load instruction has a nonzero derivative, we set the derivative of the loaded variable to 1.0 if it does not already have a nonzero derivative. This is a simplifying approximation that may lead errors in evaluating the proximal gradient. However, we note that proximal derivatives on load operations can potentially be evaluated by sampling adjacent memory locations. We leave this to future work.

### 3.4 Program Gradient as Dataflow

To use gradients as a measure of dataflow, we compute gradient between a set of user defined sources and sinks. We set the initial vectors in \( \mathbf{V} \) so that each vector is all 0s except for an initial derivative on each source of +1 or −1. We then execute the program and propagate the derivatives over each operation with the chain rule and derivatives defined in Algorithm 1. While the program is executing we record derivatives at each sink, and accumulate the gradient on each sink from all the sources. Cumulatively, the gradients on all sinks form the Jacobian \( \mathbf{J} \) between sources and sinks.

Algorithm 2 formally describes the process for computing the gradients from a set of sources to each designated sink in program. The returned Jacobian \( \mathbf{J} \) contains the gradients of each sink based on the largest derivative propagated to it from each source (sinks may record multiple derivatives from a single source if, for example, the sink is in a loop).
Algorithm 2 Program Gradient Evaluation.

| Input: | \( P \) ← program under analysis |
|        | \( x \) ← program input |
|        | \( Sources \) ← \( n \) dataflow sources |
|        | \( Sinks \) ← \( m \) dataflow sinks |

1: initialize \( V \) to empty set \( \{\} \)
2: initialize \( J \) \( n \times m \) matrix to 0s
3: for \( src \) in \( Sources \) do
4: \( v^+_{src} \) ← \( \lfloor dsrc = 1, otherwise 0 \rfloor \)
5: \( v^-_{src} \) ← \( \lfloor dsrc = -1, otherwise 0 \rfloor \)
6: add \( v^+_{src} \) and \( v^-_{src} \)
7: end for
8: Execute \( P \) on input \( x \), tracking \( P'(x; V) \)
9: for \( sink \) in \( Sinks \) do
10: for each recorded \( dsink \) do
11: if \( |dsrc| > |J[src, sink]| \) then
12: \( J[src, sink] \) ← \( dsink \)
13: end if
14: end for
15: end for
16: return \( J \)

4 Implementation

We implement PGA as a new sanitizer in the LLVM framework [27] called Gradient Sanitizer (grsan). We use LLVM because it allows us to instrument a program during compilation after it has been converted to LLVM’s intermediate representation. This means that grsan can be used to instrument any program written in a language supported by LLVM, and incurs lower runtime overhead than binary instrumentation frameworks such as PIN or Valgrind [1,32]. However, we note that PGA could also be implemented in a binary instrumentation framework to facilitate analysis in cases where source code is not available.

Overall Architecture. We base grsan on LLVM’s taint tracking implementation, DataFlowSanitizer (dfsan), which uses shadow memory to track taint labels. For each byte of application memory, there are two corresponding bytes of shadow memory that store the taint label for that byte.

We modify dfsan in the following two ways: First, we add additional metadata associated with each label that stores the gradient information, which is stored in a separate table as shown in Figure 6. Each label in the shadow memory is associated with a distinct derivative value in the gradient table. The 0 label is reserved for 0 derivative, and any shadow memory lookup on a constant or unlabeled variable returns label 0.

Second, we change the dataflow propagation rules to compute gradients over each operation. Figure 6 shows an example of how the grsan instrumentation works. Given an operation \( y=2\times x \), the instrumentation first looks up the derivative for each input, 2 and \( x \), from shadow memory. If any input has a nonzero derivative, it computes the derivative for the output \( y \) and generates a new shadow memory label by incrementing the current max label by 1. It then allocates space in the shadow memory and gradient table and stores the new label and associated derivative of \( y \).

As an additional optimization, when storing an operation’s output derivative we first compare it to the input derivatives. If the output derivative is equal to either, we apply the label of the equivalent input derivative to the output instead of generating a new label and gradient table entry. Since many operations do not change the value of the derivative (e.g. \( x = x+1 \)), this significantly reduces the number of distinct labels that need to be tracked.

In the current implementation, grsan tracks derivatives from a single source at a time, propagating the two derivatives from the source in parallel. When computing a gradient over multiple sources (e.g. bytes in an input file), we execute the program once for each source. We intend to extend grsan to support multiple sources in parallel in future work.

Gradient Propagation Instrumentation. For differentiable operations such as a floating point multiplication (fadd), grsan uses the analytical derivative of the operation. For nondifferentiable operations such as bitwise And, grsan uses an optimized version of proximal derivatives from Algorithm 1 that returns the first nonzero derivative it encounters when sampling. We found this approximation picked the same values that the proximal operator would select and is computationally lighter (i.e. does not require computing exponents).

We leave most external function calls uninstrumented, but some operations in glibc are given special instrumentation. We set the gradients for any buffer overwritten by fread or memmove to 0, and the gradients of buffers copied by memcpy or strncpy are also copied. Type cast-
<table>
<thead>
<tr>
<th>Library</th>
<th>Test Command</th>
<th>SLOC</th>
<th>File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>zlib-1.2.11</td>
<td>minigzip -d</td>
<td>3228</td>
<td>GZ/ZIP</td>
</tr>
<tr>
<td>libjpeg-9e</td>
<td>djpeg</td>
<td>8,857</td>
<td>JPEG</td>
</tr>
<tr>
<td>mupdf-1.14.0</td>
<td>mutool show</td>
<td>123,562</td>
<td>PDF</td>
</tr>
<tr>
<td>libxml2-2.9.7</td>
<td>xmllint</td>
<td>73,920</td>
<td>XML</td>
</tr>
<tr>
<td>binutils-2.30</td>
<td>objdump -xD</td>
<td>72,955</td>
<td>ELF</td>
</tr>
<tr>
<td></td>
<td>strip</td>
<td>56,330</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>52,991</td>
<td>ELF</td>
</tr>
</tbody>
</table>

Table 1: Test programs used in our evaluation.

ing instructions are handled by simply copying labels from the original value to the result.

5 Evaluation

We evaluate PGA by comparing its performance directly to DTA, and in direct applications for bug finding and security analysis. Specifically, we run experiments to answer the following questions:

1. **Dataflow Accuracy**: Is PGA more accurate than DTA in tracking dataflows?
2. **Overhead**: How does the overhead introduced by PGA compare to DTA?
3. **Guided Fuzzing**: Does using PGA to guide fuzzing lead to better edge coverage?
4. **CVE Analysis**: Can PGA detect and analyze recent CVEs that taint is typically used to detect?
5. **Bug Discovery**: Is PGA an effective tool for finding bugs?
6. **Information Leaks**: Can PGA detect and analyze memory and timing-based information leaks?

5.1 Experimental Setup

**Test Programs.** We perform tests on a set of 5 widely used file parsing libraries and 7 total programs. We use file parsers because these programs often must process files from untrusted sources, making them a common target for attacks. Table 1 shows the test programs and SLOC associated with each executable tested. In total the programs have 391,883 SLOC.

**Fuzzers Evaluated.** For our fuzzing experiments, we use the latest version of **NEUZZ**¹ and **VUzzer**².

**Test Environment.** All of our evaluations are performed on an Ubuntu 16.04 server with an Intel Xeon E5-2623 v4 2.60GHz CPU and 192G of memory unless otherwise specified.

5.2 Performance

We first evaluate the performance of PGA as a tool for dynamic dataflow analysis. In our experiments, we compare PGA to DataFlowSanitizer (dfsan), LLVM’s state-of-the-art DTA implementation. Since our implementation of PGA is based on the dfsan architecture, our setup ensures that any differences in performance between PGA and DTA are due to differences in the underlying architectures.

We compare performance in three areas: first, we estimate the accuracy of the dataflows predicted by PGA and DTA. Second, we evaluate the overhead introduced by the PGA instrumentation. Third, we compare the edge coverage achieved by a dataflow-guided fuzzer using either PGA or DTA to guide its mutation strategy.

**Evaluation Inputs.** We use the same set of initial input files for all of the performance evaluations. The gzip, pdf, and ELF files are sourced from the AFL sample seeds included in the distribution³. The jpeg input was generated from running a small jpeg image through a jpeg reduction service⁴. The libxml input was selected from the libxml² test inputs smaller than 700 bytes with the greatest AFL branch coverage.

5.2.1 Dataflow Accuracy

We evaluate the accuracy of PGA in comparison to DTA against an estimate of ground truth dataflows. This comparison setting favors DTA since it does not take the fine grained dataflow information from PGA into account (i.e., only considers binary 0/1 influence), but still illustrates the benefits of PGA’s increased precision. In addition to comparing against dfsan, we also compare against libdfdt, another widely used DTA framework that uses Intel PIN to instrument the binary directly, Neutaint, which uses the gradients of a neural network to model dataflows, and an ablation of PGA with binary gradients, grsan (binary). Notably, libdfdt tracks taint at byte level granularity and incorporates special case rules to handle operations that cancel out dataflows, such as $y = x - x$.

**Ground truth estimation.** To estimate ground truth dataflows, we measure if changes in taint sources cause changes in sink values during execution. When recording executions, we only consider executions that follow the same path to remove implicit flows, since neither DTA nor PGA can detect these. We mark each byte read from the input file as a source and each branch condition as a sink, because branches ultimately determine the behavior of a program, and because many security vulnerabilities can

---

¹www.github.com/Dongdongshe/neuzz
²www.github.com/vusec/vuzzer64
³https://github.com/google/AFL
⁴https://tinyjpg.com/
⁵https://gitlab.gnome.org/GNOME/libxml2/
Table 2: Summary of accuracy comparison results for DTA and PGA systems. Neutaint, libdft, and dfsan are state-of-the-art DTA systems, while binary grsan is an ablation of PGA that only uses binary (1 or 0) gradients to test the impact of precise gradients on accuracy. Best F1 scores for each program are highlighted. Experiments with libdft on djpeg and xmllint timed out after 24hrs. PGA (with floating point gradients) outperforms DTA on all programs, and full precision (floats) grsan outperforms binary grsan on all programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Neutaint</th>
<th>libdft</th>
<th>dfsan</th>
<th>grsan (binary)</th>
<th>grsan (floats)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>minigzip</td>
<td>0.02</td>
<td>0.55</td>
<td>0.04</td>
<td>0.42</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>djpeg</td>
<td>0.02</td>
<td>0.33</td>
<td>0.04</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mutool</td>
<td>0.002</td>
<td>0.19</td>
<td>0.004</td>
<td>0.70</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmllint</td>
<td>0.07</td>
<td>0.69</td>
<td>0.12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>objdump</td>
<td>0.03</td>
<td>0.20</td>
<td>0.05</td>
<td>0.47</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strip</td>
<td>0.02</td>
<td>0.09</td>
<td>0.03</td>
<td>0.26</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>0.06</td>
<td>0.39</td>
<td>0.11</td>
<td>0.20</td>
<td>0.59</td>
</tr>
</tbody>
</table>

Only input byte, we set the byte to 0, 255, and toggling each bit for a total of 10 samples. We found that this sampling strategy usually triggered a change in the sink variable when there was a valid dataflow.

Accuracy evaluation. We perform the accuracy evaluation on the programs shown in Table 1 using a set of small seed files (<1Kb) to make sampling each byte feasible. Since valid dataflows often only involve a few input bytes, we use F1 accuracy, which is a standard metric for evaluating predictions on imbalanced classes in classification problems. F1 accuracy is computed as \[ F1 = 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \]. Precision indicates the proportion of bytes with predicted dataflows that are correct (i.e. not false positives), while recall indicates the proportion of valid dataflows that were correctly predicted (i.e. not false negatives). Results are shown in Table 2.

Generally, PGA achieves a significant improvement in precision, achieving up a 37% increase in precision and 33% increase in F1 accuracy (20% on average) compared to the best performing DTA system, dfsan. Overall PGA gets higher F1 scores for all programs. In spite of incorporating special case dataflow cancellation rules for its bitwise and numerical operations, libdft achieves lower accuracy than dfsan in the evaluation. We hypothesize this is due to the difficulty in writing handcrafted rules for all possible X86 instructions, which leads to errors in propagation rules as noted in [12]. The binary gradient PGA ablation, grsan (binary), also has much lower accuracy than full precision PGA, indicating gradients are essential to computing accurate dataflows with PGA. We discuss the binary gradient ablation in more detail in Appendix A.

Additional Accuracy Experiments. In addition to the accuracy experiment in Table 2, we run experiments to address the following: (1) How do varying compiler optimization levels affect the accuracy of PGA vs. DTA? (2) How does PGA perform against Neutaint in Hotbyte prediction? (3) On which specific operations does PGA vary from DTA due to 0 gradients? (4) How does PGA compare with Quantitative Information Flow (QIF) techniques? We summarize the results here and describe these experiments in detail in Appendix A.

1. Compiler Optimization. PGA’s accuracy improvement over DTA is robust to varying compiler optimization levels. On average, PGA is at least 18% more accurate than DTA on compiler optimization levels -00 through -02.

2. Hotbyte Prediction. When we reproduce the Hotbyte experiment described in Neutaint [43], (i.e. identifying input bytes with the most dataflows to branches) PGA achieves 43.8% accuracy while Neutaint achieves 64.3% accuracy on average. Neutaint achieves higher average accuracy because it trains on a large corpus of recorded execution traces, while PGA and the DTA reason about a single input and execution trace at a time. We see Neutaint as a complementary method that performs well in identifying hotbytes, while PGA has better fine grained dataflow accuracy, and both methods could be used together in program analysis.

3. Zero Gradient Analysis. PGA avoids overtainting errors when it computes zero gradients on operations where DTA would propagate taint. We find the zero gradients occur most frequently on And, Remainder, Sub, Mul, and Shift operations, and that zero gradients are most often caused by masking, shifting, or composition effects.

4. QIF Comparison. We compare PGA with a QIF tool Flowcheck that quantifies information flow
Figure 7: Comparison of guided fuzzer edge coverage achieved by PGA and DTA over 100k mutations from a single seed. Overall gradient-guided fuzzing achieves up to 56% higher coverage and improves the rate of new edge discovery by 10% on average.

5.2.2 Overhead

We observe two conflicting phenomena when measuring overhead: PGA can either increase overhead due to the additional floating point storage and computation required by gradients, or decrease runtime and memory overhead when its increased precision reduces unnecessary dataflow tracking operations that use additional computation and shadow memory.

We evaluate the overhead introduced by our implementation of PGA in runtime and memory relative to dfsan on a single source dataflow. Note that if we consider overhead for multiple sources, the runtime will be lower and the memory overhead will be higher for a multi-source implementation. In the worst case PGA has 21.7% greater overhead in runtime and 21.5% in memory relative to DTA, but on average only adds 3.21% relative overhead in runtime and 1.48% in memory. Table 9 and Table 10 in Appendix B show the detailed results.

We also provide overhead measurements for libdft, although it adds significantly more overhead due to the binary instrumentation.

Result 2: On average PGA increases runtime overhead by 3.21% runtime and memory overhead by 1.48% relative to DTA, and increases runtime by 21.7% and memory usage by 21.5% relative to DTA in the worst case.

5.2.3 Dataflow-Guided Fuzzing

Since dynamic dataflow analysis is often used as a tool to guide fuzzing, we evaluate PGA in comparison to DTA as a method for guiding fuzzer mutations. Unlike our evaluation of dataflow accuracy, this experiment emphasizes the dataflow magnitude information provided by the program gradient, since bytes with the largest derivatives are selected for fuzzing.

We first compare PGA and DTA using a simple deterministic strategy for mutating input bytes based on dataflows to branches. This ensures there is no bias from randomized mutation strategies or other heuristics employed by state-of-the-art fuzzers in this evaluation. First, we execute the program with all inputs set as sources and all branches set as sinks. We then select 128 bytes from the input bytes based on the measured taint and gradient flows to branches. With PGA, the bytes with the greatest gradients are prioritized, this approach utilizes the additional information provided by PGA to improve the mutation strategy. The fuzzer performs a deterministic set of mutations on the selected 128 bytes, in which each byte in turn is set to all 256 possible values.

Enhancing state-of-the-art fuzzers. We also evalu-
We first evaluate PGA as a tool for detecting danger-
with DTA. For these CVEs, DTA overtaints on the ma-
vulnerabilities in an online manner. In the case of CVE-
out-of-memory allocation error triggered by the dataflow from an input byte, PGA directly mea-
sure the effect of input changes on the size of the allo-
can early terminate when it finds input byte
values that will trigger the out-of-memory error.

We also note the utility of the additional information
provided by gradients and how it can help distinguish vulnerab-
ity types, including stack and heap over-
by the dataflow from an input byte, PGA directly mea-

5.3 Bug Finding

Next, we show the additional information provided by
make it a useful tool for discovering and analyzing
different types of bugs in real world programs. We test
PGA against DTA in three applications: detecting and
analyzing known vulnerabilities, guiding discovery of new
bugs, and discovering information leaks.

5.3.1 Analysis of known CVEs

We first evaluate PGA as a tool for detecting danger-
ous dataflows offline in known CVEs. We instrument
the programs to mark user-controlled input as dataflow
sinks. We select 21 CVEs that cover a range
of vulnerability types, including stack and heap over-
flows, integer overflows, memory allocation errors, and
null pointer dereferences. We include CVEs from our
evaluation programs as well as openssl to demonstrate
PGA based analysis on a variety of program types.

Table 3 shows a comparison of PGA and DTA in
detecting the relevant dataflows in these CVEs. PGA
correctly identifies dataflows for 19 out of the 21 evalu-
ated CVEs, including 2 CVEs that cannot be identified
with DTA. For these CVEs, DTA overtaints on the ma-
noxious inputs and crashes due to label exhaustion, while
PGA can precisely identify the dataflows without over-
tainting. For the 2 CVEs which both PGA and DTA fail
to detect, the dataflow source indirectly propagates to
the sink through implicit dataflows (i.e. control flow).

We compare PGA and DTA as bug discovery tools by
adding additional instrumentation to record dataflows for
instruction and function arguments that can potentially
trigger program errors, such as memory allocations, copy
instructions, indexing operations, and shift operators. We
then execute the programs on a corpus of files generated
by running AFL on each program for 24 hours. Next, we
generate new inputs by changing input bytes involved
in the recorded dataflows similar to Section 5.2.3. For
PGA, we select 128 input bytes prioritized based on the
function gradient, while for DTA, we randomly select
them. We modify the values of the selected bytes based
on the gradient for PGA or by setting them to 0 or 255
for DTA.

Table 4 summarizes our results. Overall, PGA finds 22
bugs in our evaluated programs through gradient guided
modification of the inputs, including arithmetic errors,
out-of-memory allocations, and integer overflows. The
DTA guided bug search finds 15 of these 22 bugs. Of the
22 bugs, 20 have been confirmed by the developers, 3 of
them resulted in new patches, and the remaining 17 were
already patched in the latest sources of the programs.

<table>
<thead>
<tr>
<th>CVE ID</th>
<th>Vulnerability - Program</th>
<th>PGA</th>
<th>DTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2007-1657</td>
<td>stack overflow - minigzip</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-7210</td>
<td>off-by-one read - objdump</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-8396</td>
<td>heap overflow - libbfd</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-15996</td>
<td>out-of-memory - readelf</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-6543</td>
<td>integer overflow - objdump</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-6759</td>
<td>null ptr dereference - nm</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-7643</td>
<td>integer overflow - objdump</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-10372</td>
<td>heap overflow - readelf</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-11813</td>
<td>infinite loop - cjpeg</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-12698</td>
<td>out-of-memory - libiberty</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-12699</td>
<td>heap overflow - libiberty</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2020-14152</td>
<td>out-of-memory - djpeg</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-19932</td>
<td>integer overflow - strip</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-19777</td>
<td>infinite loop - mutool</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-20671</td>
<td>infinite loop - objdump</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2019-14444</td>
<td>integer overflow - readelf</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2020-1967</td>
<td>null ptr dereference - openssl</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-11212</td>
<td>divide-by-zero - cjpege</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-11214</td>
<td>heap overflow - cjpege</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2020-7041</td>
<td>invalid certificate - openssl</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2018-12697</td>
<td>null ptr dereference - libiberty</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 3: List of 21 CVEs for which the exploitable
dataflows were analyzed by PGA and DTA (dfsan and libdfst)

5.3.2 Bug Discovery

We evaluate a version of NEUZZ modified to use PGA against
unmodified NEUZZ and VUzzer, another dataflow guided
fuzzer. On average, PGA+NEUZZ improves new edge cover-
age by 12.9% over baseline NEUZZ. We hypothesize this
improvement is because the gradients produced with
PGA are more precise than the neural-network based
gradients used by NEUZZ. We discuss this experiment
and provide more detailed results in Appendix C.

Result 3: In guided fuzzing PGA increases the rate
of edge coverage growth by 10% on average compared
to DTA, and improves the edge coverage of NEUZZ, a
state-of-the-art fuzzer, by 12.9% on average.

Result 4: PGA identifies relevant dataflows in 19 out
of 21 evaluated CVEs, including 2 DTA cannot detect
due to label exhaustion. PGA and DTA both cannot
identify control-flow-based dataflows for 2 CVEs.
GRSAN_MARK_BYTE(c, 1.0); // grad = 1.0
3 cinfo->Al = (c) & 15; // grad = 1.0
4 ...
5 (*block)[natural_order[k]] =
6 (JCOEF) (v << cininfo->Al);
7 /* block[0] gradient = 8.0 */
8
9 void jpeg_idct_islow(int * block) {
10 ... int * inptr = block; // grad = 8.0
11 z2 = (int) inptr[0] * quantptr[0]
12 /* z2 gradient = 2040.0 can overflow */
13 z2 = z2 << 13;
14 /* negative z2 triggers error */
15
16 
17 /* negative z2 triggers error */
18 }

Figure 8: Arithmetic Error in djpeg.

<table>
<thead>
<tr>
<th>Library</th>
<th>Test Program</th>
<th>Integer</th>
<th>Memory</th>
<th>Corruption</th>
</tr>
</thead>
<tbody>
<tr>
<td>libjpeg-9c</td>
<td>djpeg</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>mupdf-1.14.0</td>
<td>mutool show</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>binutils-2.30</td>
<td>size</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>objdump -xD</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>strip</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Summary of new bugs found by PGA. In total there are 22 bugs found over 5 programs.

For the 7 bugs that were found by PGA and not DTA, gradient magnitude and direction allowed the search to prioritize input bytes that could trigger errors that could not be identified with DTA. We give a case study in Figure 8, which illustrates how large gradients are used to find an arithmetic error in djpeg. By altering an input byte with a large gradient to a shift operand, an overflow is triggered that results in an invalid operation. Similarly, identifying inputs with large gradients to memory operations was key to finding memory errors.

Result 5: A simple PGA guided search finds 22 bugs in the tested programs. A DTA guided search using the same strategy and inputs finds 15 of these 22 bugs.

5.3.3 Information Leak Discovery

We provide two case studies using PGA to detect side channel leaks: one example of a memory usage based side channel in objdump and an execution time based side channel in cjpeg. To detect each information leak, we marked the input file headers as sources and relevant program values as sinks, either memory allocation operands or comparison operands in loops.

In objdump, we identified a memory based side channel based on a gradient of 1 million to a malloc instruction from the ELF section header for program size. Figure 9a shows the effect of incrementing the value from 46 to 59 on the program’s total memory usage. The memory consumption is linear in the byte value if the byte is in range from 48 to 57, which can be converted to a valid number '0' to '9' in ASCII. Similarly, we identified the timing based side-channel in cjpeg by a gradient from the height field in the jpeg header to the operand of a while loop condition. Figure 9b shows the height information leak in program execution time.

Prior side channel attacks have demonstrated that these types of leaks can be exploited to learn sensitive information about a user [11, 24]. For example, one can imagine a malicious Android app that uses JPEG dimensions leaked from a browser to determine which websites the device user is visiting.

Result 6: PGA successfully detects two information leaks from file headers in objdump and cjpeg.

6 Discussion

In this section we review the implications of our results and discuss the relative advantages and limitations of PGA as an approach to dataflow analysis.

Advantages of PGA. The additional information encoded in gradients can greatly improve precision when predicting dataflows between sources and sinks (i.e. reducing the number of false positives), while the magnitude and direction information can be used to prioritize dataflows based on their significance and predicted effect. We see the benefits of the additional information from gradients in the improved performance of PGA relative to DTA in our dataflow accuracy, guided fuzzing, and vulnerability detection and analysis evaluations (Sections 5.2.1, 5.2.3, 5.3).

Limitations of PGA. While our implementation of PGA demonstrably works based on our evaluation, how to best sample non-smooth operations when evaluating proximal gradients is an open question. Our prototype uses a simple fixed sampling strategy, and does not fully implement proximal gradients on some operations, such as loads on pointers with derivatives (Section 3.3), which sometimes causes errors in the gradient evaluation. The
effect of these errors can be seen in our dataflow accuracy evaluation (Section 5.2.1), where dfsan has slightly lower recall than grsan, indicating some gradients erroneously evaluate to 0. We believe incorporating more information about specific operations in sampling strategies, as well as tracking valid domains for some operations, will reduce these errors.

A second limitation of our implementation of PGA is that, like most DTA frameworks, it does not model implicit dataflows, such as control flow dependencies. This can be seen in our CVE evaluation (Section 5.3.1), where two of our tested CVEs cannot be detected by either PGA or DTA. We intend to explore both more accurate methods for evaluating proximal gradients and modeling implicit dataflows in future work.

7 Related Work

Dynamic Taint Analysis. Dynamic Taint Analysis (DTA) tracks data flow from taint sources to taint sinks at runtime. Common applications of DTA include software vulnerability analysis and information leak detection [15, 18, 34, 58, 59]. DTA typically overestimates the tainted bytes which contributes to a large performance overhead. Therefore, much of the recent work in DTA has focused on developing more efficient systems [7, 25, 29]. Like DTA, PGA dynamically propagates dataflow information, but it provides more fine-grained information in the form of gradients. Moreover, PGA is more precise than DTA, which reduces overtainting in large programs.

Some DTA systems use bit level taint tracking to improve precision at the cost of higher overheads [54, 55]. Although we have not implemented it in our current prototype, gradients can also be propagated over individual bits based on functional Boolean analysis, and we expect it to offer similar tradeoffs in improved accuracy for higher overheads [35].

Recently, automatically learning taint rules has been used to reduce the approximation errors in DTA [12]. This approach is orthogonal to ours and could also potentially be applied to learn gradient propagation rules.

Quantitative Information Flow. Quantitative Information Flow (QIF) measures the potential transmission of information through a program using entropy based measures such as channel capacity and min-entropy [19, 28, 50]. QIF has primarily been used for detecting information leaks and ensuring the integrity of program secrets [3, 17, 23], but has also been proposed as a way of enhancing taint tracking [33]. PGA adds a different type of information as discussed in Appendix A, and does not have the high computational complexity involved in estimating information flows accurately.

Gradient-guided fuzzing. Recent fuzzers have used gradient approximations to guide their mutation process. Angora estimates finite differences, an approximation of gradients with many known limitations especially for high-dimensional problems, by executing the program on modified inputs and recording the changes in the outputs [10, 39]. NEUZZ, MTFuzz and Neutaint train neural networks to predict program branch behavior and use the network’s gradients to guide the mutation algorithm [43–45]. This incurs less overhead than instrumentation based methods but is also less exact since it operates on an approximate model of the program. By contrast, PGA computes gradients directly over the program’s individual instructions and therefore produces precise gradients.

Program Smoothing. Prior work has explored computing gradients with smooth interpretation of a program via a Gaussian kernel [8, 9] or parametric relaxation of SMT [41, 57]. These methods use symbolic reasoning and have not been applied to analysis of real world programs. PGA’s approximation methods are more efficient and have been successfully demonstrated on real world programs.

8 Conclusion

In this paper we introduce proximal gradient analysis (PGA), a novel theoretically-grounded approach to dataflow analysis that uses non-smooth calculus techniques to compute gradients over programs. PGA is more precise than dynamic taint tracking and provides more fine-grained information about program behavior. We provide a prototype implementation of PGA based on the LLVM framework and show that it outperforms three state-of-the-art DTA systems in accuracy while adding less than 5% overhead on average. Finally, we show PGA is an effective tool for security analysis, identifying relevant dataflows for 19 different CVEs, discovering 22 bugs, and detecting 2 side-channel leaks in 7 real world programs. We hope that our approach to program analysis will motivate other researchers to explore new techniques exploiting the rich non-smooth analysis literature.

Acknowledgements

We thank our shepherd Lujo Bauer and the anonymous reviewers for their constructive and valuable feedback. The first author is supported by an NDSEG Fellowship, and the second author is supported by an NSF Graduate Fellowship. This work is sponsored in part by NSF grants CNS-18-42456, CNS-18-01426, CNS-16-17670; ONR grant N00014-17-1-2010; an ARL Young Investigator (YIP) award; a NSF CAREER award; a Google Faculty Fellowship; and a Capital One Research Grant, as well as European Union Marie Sklodowska-
Curie grant agreement 690972 (PROTASIS) and innovation programme under grant agreement No. 786669 (ReAct). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, European Union, ONR, ARL, NSF, Google, or Capital One.

References


[42] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&F)*, pages 317–331. IEEE, 2010.


<table>
<thead>
<tr>
<th></th>
<th>Opt 0</th>
<th>Opt 1</th>
<th>Opt 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dfsan</td>
<td>gscan</td>
<td>dfsan</td>
</tr>
<tr>
<td>minigzip</td>
<td>0.39</td>
<td>0.57</td>
<td>0.42</td>
</tr>
<tr>
<td>djpeg</td>
<td>0.36</td>
<td>0.69</td>
<td>0.29</td>
</tr>
<tr>
<td>mutool</td>
<td>0.62</td>
<td>0.64</td>
<td>0.56</td>
</tr>
<tr>
<td>xmllint</td>
<td>0.76</td>
<td>0.92</td>
<td>0.73</td>
</tr>
<tr>
<td>objdump</td>
<td>0.52</td>
<td>0.71</td>
<td>0.48</td>
</tr>
<tr>
<td>strip</td>
<td>0.33</td>
<td>0.63</td>
<td>0.31</td>
</tr>
<tr>
<td>size</td>
<td>0.53</td>
<td>0.74</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Table 5: Effect compiler optimization levels on dataflow F1 accuracy. The table shows gscan has significantly higher F1 accuracy than dfsan for all three measured optimization levels (> 18% average)
A Additional Accuracy Experiments

We describe the additional accuracy evaluations summarized in Section 5.2.1 here. Specifically, the ablation of gradient information, the effects of compiler optimization, PGA vs. Neutaint in coarse grained dataflow prediction, analysis of 0 gradients, and a comparison with QIF.

Gradient Ablation. We measure the effect of the gradient information on determining accurate dataflows by performing an ablation with binary valued gradients. The ablation uses the same proximal gradient propagation rules, but rounds all gradients to 0 or 1. In effect, this converts PGA into DTA with PGA propagation.

Results of the comparison are shown in Table 2. PGA with floating point gradient information performs significantly better than PGA with binary gradients for every program. These results indicate that precise gradients are key to the performance gains achieved by PGA because they compose accurately over multiple operations.

Compiler Optimization. We evaluate the impact of compiler optimization levels on dataflow accuracy at 3 optimization levels: -00, -01, and -02. Table 5 summarizes the effects of 3 compiler optimization levels on dataflow F1 accuracy. Increasing the compiler optimization levels reduces the accuracy of both PGA and DTA by a small amount (<3.6%) for both -01 and -02. On average, PGA is at least 18% more accurate than DTA for all three tested optimization levels.

Neutaint hotbyte evaluation. Neutaint’s neural network based approach does not perform well in fine grained dataflow prediction, but is better suited to identifying hot bytes (input bytes that are most influential to program behavior). We therefore perform the hot byte evaluation described in [43] on PGA. Our results are summarized in Table 6. On average, PGA predicts hot bytes with 43.75% accuracy, while Neutaint predicts hot bytes with 64.25% accuracy. We see Neutaint as a complementary method to PGA, where PGA is better suited to fine grained dataflow prediction and both methods could be used together in program analysis.

Zero gradient analysis. PGA is able to avoid over-tainting when it computes a zero gradient on an instruction DTA would mark as tainted. Therefore we investigate the distribution of zero gradients across programs and instruction types to determine where and how PGA is more precise than DTA. For each program and each type of instruction, we count how many times the instruction had zero gradient in the execution traces from the accuracy evaluation. Table 8 shows the results of this analysis for each instruction and program.

QIF Comparison. We compare PGA with the latest version of a publicly available QIF tool Flowcheck [28]. We perform a similar experiment to Section 5.2.1, but since Flowcheck does not byte-level granularity, we compute accuracy by aggregating flows over all bytes so that PGA is not unfairly advantaged. We outperform Flowcheck in terms of F1 accuracy by 22% on average on all of the evaluated programs as summarized in Table 7.

B Runtime and Memory Overhead Evaluation

Program Overhead. We evaluate the overhead introduced by our implementation of PGA in runtime and memory and compare it to dfsan for a single taint/gradient source. To measure overhead, we execute each program while recording runtime and memory usage. For runtime we perform 5,000 executions for each measurement. We perform each measurement 5 times and average the measured runtime and memory usage.

Tables 9 and 10 detail the runtime and memory overhead per program in our evaluation. In the worst case PGA has 21.7% greater overhead in runtime and 21.5% in memory relative to DTA, but on average only adds 3.21 % relative overhead in runtime and 1.48% in memory. We also provide overhead measurements for libdft, although it adds significantly more overhead due to the binary instrumentation.
### Program Summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Instrs</th>
<th>%Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>minigzip</td>
<td>3012</td>
<td>28.2%</td>
</tr>
<tr>
<td>djpeg</td>
<td>703</td>
<td>38.7%</td>
</tr>
<tr>
<td>mutool</td>
<td>401</td>
<td>40.4%</td>
</tr>
<tr>
<td>xmllint</td>
<td>430</td>
<td>39.5%</td>
</tr>
<tr>
<td>objdump</td>
<td>1070</td>
<td>39.0%</td>
</tr>
<tr>
<td>strip</td>
<td>3089</td>
<td>41.0%</td>
</tr>
<tr>
<td>size</td>
<td>659</td>
<td>19.3%</td>
</tr>
</tbody>
</table>

### Instruction Summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Total</th>
<th>%Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>6756</td>
<td>30.2%</td>
</tr>
<tr>
<td>URem</td>
<td>214</td>
<td>29.0%</td>
</tr>
<tr>
<td>Sub</td>
<td>1214</td>
<td>21.0%</td>
</tr>
<tr>
<td>Mul</td>
<td>875</td>
<td>15.9%</td>
</tr>
<tr>
<td>LShr</td>
<td>2377</td>
<td>14.4%</td>
</tr>
<tr>
<td>AShr</td>
<td>149</td>
<td>6.0%</td>
</tr>
<tr>
<td>Add</td>
<td>895</td>
<td>5.7%</td>
</tr>
</tbody>
</table>

Table 8: Analysis of operations from execution traces where gradient drops to 0, aggregated for each program and for each type of instruction across all programs. Outputs of these operations will have 0 gradient but still be marked as tainted by DTA.

### Table 9: Program runtime overhead measurements averaged over five runs for a single taint/gradient source. Libdft overhead is measured relative to running a program only with PIN. Dfsan and Grsan are measured relative to uninstrumented programs. After 6 hours, libdft execution timed out on djpeg.

<table>
<thead>
<tr>
<th>Program</th>
<th>Overhead</th>
<th>Overhead</th>
<th>Overhead</th>
<th>rel. to dfsan</th>
</tr>
</thead>
<tbody>
<tr>
<td>libdft</td>
<td>2379.5%</td>
<td>54.7%</td>
<td>61.5%</td>
<td>4.4%</td>
</tr>
<tr>
<td>dfsan</td>
<td>-</td>
<td>70.5%</td>
<td>73.7%</td>
<td>1.9%</td>
</tr>
<tr>
<td>grsan</td>
<td>853.5%</td>
<td>198.4%</td>
<td>262.1%</td>
<td>21.5%</td>
</tr>
<tr>
<td>grsan rel. to dfsan</td>
<td>231.4%</td>
<td>5.5%</td>
<td>0.0%</td>
<td>-5.2%</td>
</tr>
<tr>
<td>size</td>
<td>152.5%</td>
<td>101.1%</td>
<td>107.1%</td>
<td>3.0%</td>
</tr>
<tr>
<td>objdump</td>
<td>180.0%</td>
<td>133.2%</td>
<td>131.2%</td>
<td>-0.9%</td>
</tr>
<tr>
<td>strip</td>
<td>142.5%</td>
<td>12.0%</td>
<td>11.4%</td>
<td>-2.2%</td>
</tr>
</tbody>
</table>

### Table 10: Memory overhead for each program averaged over five runs relative to uninstrumented programs for a single taint/gradient source. Grsan may increase or decrease overhead because gradients require more memory to store, but may use less overall memory due to increased precision. On average, grsan adds 1.48% additional overhead relative to dfsan.

<table>
<thead>
<tr>
<th>Program</th>
<th>dfsan</th>
<th>grsan</th>
<th>grsan rel. to dfsan</th>
</tr>
</thead>
<tbody>
<tr>
<td>minigzip</td>
<td>183.7%</td>
<td>245.3%</td>
<td>21.7%</td>
</tr>
<tr>
<td>djpeg</td>
<td>274.6%</td>
<td>291.9%</td>
<td>4.1%</td>
</tr>
<tr>
<td>mupdf</td>
<td>112.4%</td>
<td>124.7%</td>
<td>5.8%</td>
</tr>
<tr>
<td>xmllint</td>
<td>346.6%</td>
<td>258.5%</td>
<td>-19.7%</td>
</tr>
<tr>
<td>size</td>
<td>373.3%</td>
<td>392.4%</td>
<td>4.0%</td>
</tr>
<tr>
<td>objdump</td>
<td>345.6%</td>
<td>323.5%</td>
<td>-5.0%</td>
</tr>
<tr>
<td>strip</td>
<td>344.5%</td>
<td>342.1%</td>
<td>-0.5%</td>
</tr>
</tbody>
</table>

### Table 11: New edge coverage for each program over 24 hours by three different fuzzers. Vùzzer encounters an error in its taint tracking on minigzip and crashes. Overall, PGA+NEUZZ improves NEUZZ edge coverage on average by 12.9%. Note that our results are slightly different from the original NEUZZ and Vùzzer results due to differences in test environments, input corpuses, and program versions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Vùzzer</th>
<th>NEUZZ</th>
<th>PGA + NEUZZ</th>
<th>PGA+NEUZZ rel. to NEUZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>minigzip</td>
<td>-</td>
<td>87</td>
<td>94</td>
<td>8.1%</td>
</tr>
<tr>
<td>djpeg</td>
<td>7</td>
<td>645</td>
<td>686</td>
<td>6.4%</td>
</tr>
<tr>
<td>mupdf</td>
<td>156</td>
<td>376</td>
<td>430</td>
<td>14.4%</td>
</tr>
<tr>
<td>xmllint</td>
<td>282</td>
<td>957</td>
<td>1079</td>
<td>12.8%</td>
</tr>
<tr>
<td>size</td>
<td>474</td>
<td>1580</td>
<td>2064</td>
<td>30.6%</td>
</tr>
<tr>
<td>objdump</td>
<td>247</td>
<td>1813</td>
<td>2014</td>
<td>11.1%</td>
</tr>
<tr>
<td>strip</td>
<td>1337</td>
<td>3394</td>
<td>3637</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

### C Evaluation on Current Fuzzers

We also evaluate if the gradient information from PGA can improve the performance of state-of-the-art fuzzers such as NEUZZ and Vùzzer. We use NEUZZ as a basis because it has higher edge coverage as seen in Table 11 and already incorporates gradients from a neural network in its mutation strategy. We modify NEUZZ so that it uses the PGA gradients to guide its mutation strategy. We run grsan on its inputs and send the resulting gradients to the NEUZZ backend. Note that NEUZZ is designed to operate on gradients, so we did not modify it to also use DTA. We provide a controlled comparison of PGA vs. DTA for guided fuzzing in Evaluation 5.2.3.

We compare the additional edge coverage achieved by the fuzzers over a 24hr run. Since we use some programs with different file formats from the original NEUZZ benchmark, we use a new seed corpus generated by running AFL on each program for 1 hour. We perform this experiment using cloud hosted virtual machines. Table 11 summarizes the modified PGA+NEUZZ against baseline NEUZZ and Vùzzer. On average, PGA+NEUZZ improves new edge coverage by 12.9% over baseline NEUZZ. We hypothesize this improvement is because the gradients produced with PGA are more precise than the neural-network based gradients used by NEUZZ. The very similar results in edge coverage on minigzip are caused by the CRC check in minigzip, which causes the program to exit early on most new inputs. Vùzzer crashes on minigzip due to an error in its taint tracking and achieves a low edge coverage for djpeg because of the high overhead of PIN’s dynamic binary instrumentation for taint tracking. Note that our results are slightly different from the original NEUZZ and Vùzzer results due to different initial seed corpuses, program versions, and test environments.