Poster: Precise Dynamic Dataflow Tracking with Proximal Gradients

Gabriel Ryan, Abhishek Shah, Dongdong She, Suman Jana
Columbia University
New York, USA
{gabe, dongdong, suman}@cs.columbia.edu, abhishek.shah@columbia.edu

Koustubha Bhat
Vrije Universiteit
Amsterdam, Netherlands
k.bhat@vu.nl

***Abstract*—Dataflow analysis is a fundamental technique in the development of secure software. It has multiple security applications in detecting attacks, searching for vulnerabilities, and identifying privacy violations. Taint tracking is a type of dataflow analysis is that tracks dataflow between a set of specified sources and sinks. However, taint tracking suffers from high false positives/negatives due to fundamentally imprecise propogation rules, which limits its utility in real world applications.**

We introduce a novel form of dynamic dataflow analysis, called proximal gradient analysis (PGA), that not only provides much more precise dataflow information than taint tracking, but also more fine grained information dataflow behavior in the form of a gradient. PGA uses proximal gradients to estimate derivatives on program operations that are not numerically differentiable, making it possible to propogate gradient estimates through a program in the same way taint tracking propogates labels. By using gradient to track dataflows, PGA naturally avoids many of the propogation errors that occur in taint tracking. We evaluate PGA on 7 widely used programs and show it achieves up to 39% better precision that than taint while incurring lower average overhead due to the increased precision.

***Index Terms*—poster, taint, dataflow, program analysis, nonsmooth optimization, gradient**

I. Introduction

Dataflow analysis is a fundamental technique in the development of secure software. It has multiple security applications in detecting attacks, searching for vulnerabilities, and identifying privacy violations [1], [5]. One of the most effective techniques of dataflow analysis is taint tracking, which tracks which internal variables are affected by the input [3]. However, taint tracking suffers from high false positives/negatives due to fundamentally imprecise propogation rules, which limits its utility in real world applications.

We introduce a novel form of dynamic program analysis, called proximal gradient analysis (PGA), that not only provides much more precise dataflow information than taint tracking, but also more overall information about program behavior in the form of a gradient. PGA uses proximal gradients to estimate derivatives on program operations that are not numerically differentiable, making it possible to propogate gradient estimates through a program in the same way taint tracking propogates labels [4]. By using gradient to track dataflows, PGA naturally avoids many of the problems with over approximation that occur in taint tracking.

Figure 2 gives an example of an operation on which PGA provides more precise and fine grained dataflow information

```
taint source: x
taint sink: y
// x is a 4 byte int
int x = 0x12345678;
for (int i=0; i<6; i++){
    y[i] = x;
    x = x<<8;
}
```

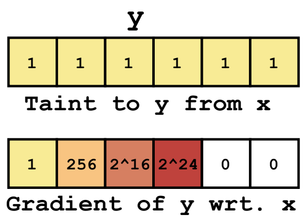

Fig. 1. **Example of a program in which an iterated shift operation on an integer will cause over-tainting, while gradient will precisely identify how the source variable (x) influences the sink(y). Deeper shades of red indicate greater degrees of influence.**

compared to taint tracking. The source integer x is left shifted by a byte every iteration of the for loop and then assigned to a position in the sink array y. After the first 4 iterations, all the bytes of x's initial value have been shifted out and x goes to 0. At this point, there is no dataflow between x and the value of y[i], since it will always be 0. PGA correctly identifies this, and also identifies that changes in x have a much larger effect on higher indexes in the array y. In contrast, taint tracking will mark all of the integers in y with x's label.

II. Background

Our approach to program analysis draws on work in three fields: Dyanmic Dataflow Analyis, Nonsmooth Optimization, and Automatic Differentiation. Dynamic Dataflow Analysis models the flow of data through a program by tracking variable interactions and has applications in both compiler optimization and detection of security vulnerabilities, but suffers from high false positive rates that limit its utility. Nonsmooth Gradient Approximation involves a collection of methods that have been developed in the field of Nonsmooth Optimization for approximating gradients in cases where the gradient cannot be evaluated analytically. These methods make it possible to approximate gradients on discrete and nonsmooth functions in a principled way based on the local behavior of the function. Finally, we draw on the field of Automatic Differentiation, which involves methods for computing gradients over programs compused of semi smooth numerical operations, but not general programs with discrete and nonsmooth operations.

To evaluate gradients over nonsmooth operations, we use a method from the discrete optimization literature called proximal gradients [4]. Proximal gradients use the minimum point within a soft bounded region. This region is defined by a cost function that increases quadratically with distance from the evaluation point. Proximal gradients use a specialed operator, called the proximal operator, which is defined as follows when evaluated on a given point $\bar{x}$:

$$prox_f\left(\bar{x}\right) = argmin_x\left(f\left(x\right) + \tfrac{1}{2}||x - \bar{x}||_2^2\right) \qquad (1)$$

The notation $argmin_x$ indicates that the operator selects the value of x that minimizes both value of function $f\left(x\right)$ and the distance cost $\left(\frac{1}{2}\right)||x - \bar{x}||_2^2$ that increases quadratically with the distance of x from $\bar{x}$. Evaluating the proximal operator will give the minimum point near the point at which it is evaluated. This point can then be used to compute the largest directional derivative in the region near the point.

$$prox_{\nabla f}\left(\bar{x}\right) = \frac{f\left(\bar{x}\right) - f\left(prox_f\left(\bar{x}\right)\right)}{\bar{x} - prox_f\left(\bar{x}\right)} \qquad (2)$$

III. Methodology and Implementation

We implement PGA as a new type of code sanitizer in the LLVM Framework. LLVM is a compiler framework that uses an Intermediate Representation that resembles high level assembly for instrumentation and optimization [2]. Adding instrumentation at the IR level allows it to be compiled into the binary, making it significantly faster than instrumenting the binary directly, and allowing it to operate on programs written in any language supported by LLVM.

Our implementation is based on the dynamic taint tracking sanitizer in LLVM, known as DataFlowSanitizer or dfsan. For each byte of application memory, dfsan has two corresponding bytes of shadow memory that store the label for that byte. In order to track gradients, we modify dfsan to store a gradient associated with each shadow label in a separate table. Every operation is instrumented to evaluate its partial derivative and generate a new label.

IV. Evaluation

We perform tests on a set 5 widely used file parsing libraries and 7 total programs, zlib, libxml, libjpeg, mupdf, and readelf, objdump, and strip in binutils.

We evaluate the precision of PGA in comparison to DTA against an estimate of ground truth dataflows. To estimate ground truth, we mark bytes read from the input file as sources and branch conditions as sinks, and execute the program while modifying each byte in the input to determine which bytes effect each branch condition. We focus on branch constraints because they determine the behavior of a program, and because many security vulnerabilities in a program can only be exploited when certain branches are taken. We generate sample inputs by setting each byet 0, 255, and toggling each bit.

Results for this experiment are shown in table I. PGA achieves as much as a 39% increase in precision and has better f1 scores for all tested programs except `mutool` and `xmllint`, which are equal. We also evaluate the runtime of instrumented programs on the same inputs and find that on average PGA has less than 5% more overhead than taint tracking, and in the worst case was 20% greater.

	Taint			Gradient		
Program	Precision	Recall	F1	Precision	Recall	F1
minigzip	0.55	0.86	0.68	0.94	0.71	**0.81**
djpeg	0.63	0.73	0.68	0.96	0.61	**0.74**
mutool	1.0	0.01	**0.02**	1.0	0.01	**0.02**
xmllint	0.97	0.39	**0.56**	0.97	0.39	**0.56**
readelf	0.17	0.95	0.28	0.18	0.93	**0.30**
objdump	0.77	0.80	0.78	0.94	0.79	**0.85**
strip	0.60	0.83	0.70	0.88	0.79	**0.84**

TABLE I
Summary of precision comparison results for taint and gradient analysis. Best F1 scores for each program are highlighted. PGA outperforms DTA on all programs except mutool and xmllint, which are tied.

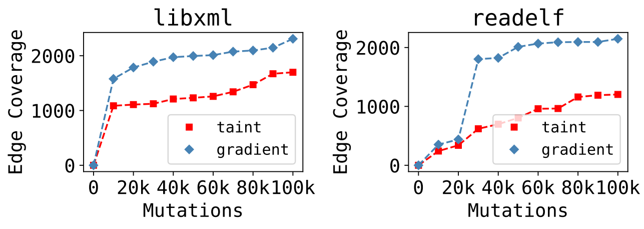

Fig. 2. **Comparison of gradient and taint guided fuzzing on `libxml` and `readelf`. Gradient guided fuzzing achieves 0.57% greater edge coverage on average after 100k mutations.**

In addition to evaluated against estimated ground truth dataflows, we also show PGA is an effective tool for guiding mutation in fuzzers. We select bytes to be mutated based on the magnitude of their derivates to branch conditions, and compare to bytes selected with taint tracking, which must be sampled randomly since taint does not distinguish degrees of influence between between variables.

V. Conclusion

We introduce a new type of Dynamic Dataflow Analysis, called Proximal Gradient Analysis, and show it outperforms Taint Tracking as a predictor of which inputs effect branch variables by up to 39% over 7 test programs with comparable overhead, as well as . We are currently investigating applications of Gradient Analysis to vulnerability discovery and information leak analysis.

References

[1] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[2] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[3] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[4] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014.

[5] 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 *2010 IEEE Symposium on Security and Privacy*, pages 317–331. IEEE, 2010.