Proximal Gradient Analysis for Vulnerability Detection and Defense

```
#define MAX_SIZE 2000
#define MAX_OBJ_SIZE 5 // programmer error: not applied!

void* alloc_input (unsigned int* input) {
        if (input[0] > MAX_SIZE || input[1] > MAX_SIZE)
                return 0;

        size_t size = input[0], obj_size = 2<<input[1];
        return malloc(size * obj_size);
}
```

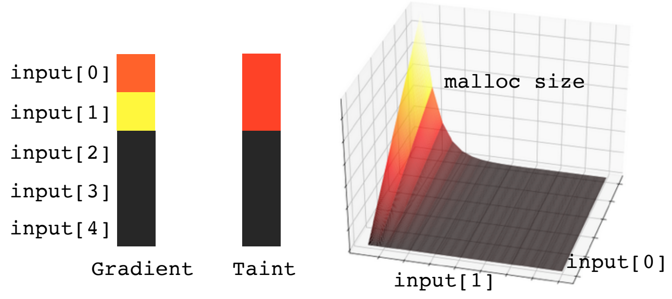

Figure 1: Comparison of taint and Gradient Analysis on a memory allocation.

So far this year, 15,354 vulnerabilities have been reported on publicly deployed software and systems. This is already more than the total reported in 2017 and more than twice the total of 6,441 in 2016 [1]. This drastic increase in reported vulnerabilities reflects the fact that developing secure software and systems is a fundamentally difficult and unsolved problem. However, even as the number of vulnerabilities increases, the danger posed by each vulnerability is becoming more acute as citizens, businesses and government agencies move more information online and make greater use of networked devices.

Dynamic program analysis is a fundamental technique in the development of secure software. It involves instrumenting programs to analyze run-time behavior and has multiple applications in vulnerability detection, malware analysis, and automatic test case generation [2, 3, 4, 5]. One of the most effective techniques in dynamic program analysis is Taint Analysis, which tracks which internal variables are affected by the input [6]. However, Taint Analysis has two significant limitations that reduce its utility—it gives no information about how the inputs it tracks affect tainted variables, and it is prone to over-approximation. For example, in the function shown in figure 1, `alloc_input`, a Taint Analysis would identify that the first two integers of input affect the malloc but give no indication that small changes to the second input could potentially trigger an error. Meanwhile, in statements like `y = x – x;` Taint Analysis marks the output as tainted, even though the value of `x` clearly has no effect on `y`. Since the search space for vulnerabilities grows exponentially with the number of inputs involved, these over-approximations limit the applicability of taint analysis to larger programs.

To address these limitations, I have recently developed a new alternative to Taint Analysis called Proximal Gradient Analysis, or simply Gradient Analysis. Gradient Analysis works by propagating gradients through a program, similarly to how a neural network trains layer by layer, estimating the gradient of each operation within a program with regard to its inputs. Computing gradients over a program is challenging due to branching behavior and other nonsmooth operations, but by applying nonsmooth optimization methods such as smoothing and principled sampling with proximal gradients, it is possible to accurately estimate program gradients [7, 8, 9].

Gradient Analysis addresses both limitations of Taint Analysis as follows:

i. Gradients provide additional information about how changes to input affect program behavior, enabling the user to deliberately alter the input to trigger desired behaviors. The `alloc_input` function shows how this method could be used to identify which variables are most significant and hone in on error triggering inputs.

ii. Gradients are not prone to over-approximation. In cases like the statement `y = x – x;` the gradient is naturally 0 where variables do not affect each other, regardless of if they are in the same operation. This makes gradients a more precise tool for tracking data flow through a program.

I have already demonstrated that Gradient Analysis is a promising research direction by developing a prototype compiler plugin and conducting preliminary experiments. These experiments show Gradient Analysis is more precise than Taint Analysis and that the plugin accurately computes gradients on a widely used compression library (zlib) and several test programs. The compiler plugin, which uses the LLVM framework, computes gradients for numerical operations in source function calls, simple reads and writes, and bitwise integer operators [9]. However, development of a robust and scalable Gradient Analysis framework will require extensive development and experimental work.

Objective: I propose development of a scalable Proximal Gradient Analysis framework with application systems for targeted vulnerability detection and guided fuzzing.

Aim 1: Development of a scalable Proximal Gradient Analysis framework using robust nonsmooth optimization methods for gradient approximation. This stage of the project will have two primary objectives: 1) engineering the gradient analysis instrumentation to support more operations while maintaining scalability and 2) experimentally determining the best approximation methods and their parameters for estimating gradient on nonsmooth operations.

The current framework will be extended to estimate gradients on operations that are not currently supported, such as function calls on uninstrumented external library functions, indexing into memory, and indirect dataflow effects through branching and loops. In addition, framework will profiled, optimized, and stress-tested on large real world programs, such as Apache Server and Mysql Database. To perform this optimization, I will leverage previous work on developing scalable Taint Analysis frameworks such as Libdft and Minimu [11, 12].

The operations that will be added to the framework have characteristics that make evaluating the gradient challenging but can be addressed with appropriate methods in optimization. These operations are often discrete, nonsmooth, and in many cases do not have analytical derivatives. There are two possible approaches to handling these operations. One approach involves sampling with proximal operators, which evaluate the local minima within a region soft bounded by the squared L2 norm [9]. Using a vector to a minima within a region as a proxy allows the operator to estimate gradient when discreteness and nonsmoothness make it impossible to evaluate directly. The region proximal operator samples will be bounded by tracking Lipschitz bounds on functions, meaning that within a region their maximum rate of change is bounded. In practice, fully sampling a region for each nonsmooth operator will be too computationally intensive in most cases, so the degree of sampling necessary to achieve a good balance between accuracy and performance will be determined experimentally.

The second approach is to generate differentiable approximations of programs via smoothing and relaxation. These involve creating smooth approximations of discrete, nonsmooth functions via kernel methods [8]. For these methods, the degree of smoothing can be varied based on input gradient so that functions whose inputs are likely to vary greatly will take a wider range of possible outputs into account. Which smoothing method is most effective and what degree of smoothing is necessary to accurately reflect program behavior will be experimentally determined.

In addition to smoothing or sampling, more complex approaches may be needed to develop meaningful approximations of control flow structures like branches and loops. Static dataflow analysis can be used to determine if a program variable can potentially be affected by a branch, which can then be sampled to determine the gradient of that variable with regard to the branch conditions.

Overall, the goal of Aim 1 is to provide a framework that is scalable, robust, and can accurately estimate gradients for a wide range of program behaviors. This framework will not only be useful for my subsequent research, but also will be shared as an open source tool that will be valuable to the program analysis community.

Aim 2: Develop targeted vulnerability detection system using Gradient Analysis. A program under test will be instrumented to track the gradients of potentially vulnerable functions using the LLVM framework. Targeted functions will include syscalls as well as methods that can trigger buffer overflows and other errors such as strcpy and integer division. After running the instrumented program on an input, the system will select a single vulnerable function with a nonzero gradient on its parameters, indicating how some parts of the input affect these parameters, and will attempt to change those parameters in a way that triggers an error. By modifying the inputs either towards or away from the gradient, the system can control how internal values change. For example, it will attempt to maximize the size of a memory allocation or set the denominator in a division to 0. This process is similar to taint guided fuzzing but will involve picking inputs in a more deterministic manner.

The process of maximizing or minimizing discrete values based on gradients is a nonsmooth optimization problem. There are several possible strategies for solving these types of problems. The simplest of these is the subgradient method, which simply follows the maximum subgradient on each step until a local maxima or minima is reached [13]. However, evaluating the gradient on each step would be expensive, so an alternative is to evaluate the gradient once and perform a line search in the direction of the gradient [14]. In practice, a combination of multiple gradient evaluations and line search will likely be most effective, and the tradeoffs between optimization accuracy and computational performance involved will have to be evaluated experimentally. Since program behavior is often nonconvex, other methods such as simulated annealing that improve optimization robustness will also be evaluated [15].

Like Taint Analysis, Gradient Analysis involves tracking dataflow through the entire program and instrumenting every operation, which results in a significant performance penalty. To achieve a high level of throughput from the system, most executions will be performed on a second instance of the program under test, which will only be minimally instrumented to track branch coverage and the values input to vulnerable functions. Thus, the system will only perform slower gradient analysis executions periodically,using the gradient information to perform many executions with inputs designed to exercise the target function.

A run of the system on a program will be performed as follows: Initially, the system will run a set of seed inputs, which may be picked manually or generated via another test technique such as fuzzing. Whenever an input that exercises a new portion of the code is discovered, it will be run on the gradient instrumented version of the program, and the gradients of each potential vulnerability will be saved. The system will then iterate through the gradient of each vulnerable function and perform constrained maximization or minimization on it, attempting to trigger an error if is possible to do so.

Aim 3: Achieve faster branch coverage during fuzzing via optimization of branch input gradients. In addition to targeting specific types of bugs, gradient analysis can be used to perform baseline fuzzer performance in terms of branch coverage. The gradients of parameters on branches can be used to guide changes to the program input to satisfy particular branch conditions.

To incorporate gradient based branch coverage into the framework, the fuzzer strategy will be modified as follows: On an initial run, the gradient of each branch with regard to the input will be recorded, and a new branch will selected to target for exploration starting with the an unvisited branch in the trace. The input will then be optimized to satisfy the selected branch constraint, using the same nonsmooth optimization methods as targeted bug search. After the new branch is explored, it will select another unvisited branch as a target. One possible problem that may arise is that the gradient optimization may not always be able to satisfy branch constraints, either because they are impossible to satisfy or because the branch parameters have highly nonconvex behavior that is difficult to optimize. Difficult optimizations can be mitigated to a certain extent with techniques such as random restarts on new inputs that also visit the branch, but a timeout will also be needed to ensure the fuzzer does not waste too much time on branch conditions that are difficult or impossible to satisfy. How much time is allowed per optimization, and whether more complex scheduling strategies such as revisiting branches with iteratively longer timeouts are worthwhile will need to be determined experimentally.

Contributions: Gradient Analysis is a fundamentally new technique for program analysis with immediate national security applications in hardening cyber infrastructure and systems against attack. In addition to the immediate applications in vulnerability detection and guided fuzzing detailed in this proposal, it can be used in any application where Taint Analysis is currently used and give a far more detailed and precise picture of program behavior. However, the potential applications go far beyond program analysis. By making a program differentiable, it becomes possible to use backpropagation to train a program – with a gradient computed for every operation, parameters within a program can potentially be trained using a set of inputs and an objective function defined on the program outputs. This could be used, for example, to train filters in a program to block attacks while still allowing all regular inputs. Finally, Gradient Analysis could be used in hybrid neural network programs, in which a neural network could have a differential program embedded within it, enabling the network to perform operations that cannot be expressed in regular tensor operations.

References:

[1] "Current CVSS Score Distribution For All Vulnerabilities." *CVE Details*, 1 Dec. 2018, www.cvedetails.com/.

[2] Newsome, James, and Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software." *NDSS*. Vol. 5. 2005.

[3] Wang, Tielei, et al. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection." *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010.

[4] Moser, Andreas, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis." *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007.

[5] Wassermann, Gary, et al. "Dynamic test input generation for web applications." *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008.

[6] Schwartz, Edward J., 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)." *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010.

[7] Rockafellar, R. Tyrrell, and Roger J-B. Wets. *Variational analysis*. Vol. 317. Springer Science & Business Media, 2009.

[8] Wand, Matt P., and M. Chris Jones. *Kernel smoothing*. Chapman and Hall/CRC, 1994.

[9] Parikh, Neal, and Stephen Boyd. "Proximal algorithms." *Foundations and Trends® in Optimization* 1.3 (2014): 127-239.

[10] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[11] Kemerlis, Vasileios P., et al. "libdft: Practical dynamic data flow tracking for commodity systems." *Acm Sigplan Notices*. Vol. 47. No. 7. ACM, 2012.

[12] Bosman, Erik, Asia Slowinska, and Herbert Bos. "Minemu: The world's fastest taint tracker." *International Workshop on Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, 2011.

[13] Kiwiel, Krzysztof Czesław. "An aggregate subgradient method for nonsmooth convex minimization." *Mathematical Programming* 27.3 (1983): 320-341.

[14] Lewis, Adrian S., and Michael L. Overton. "Nonsmooth optimization via quasi-Newton methods." *Mathematical Programming* 141.1-2 (2013): 135-163.

[15] Szu, Harold H., and Ralph L. Hartley. "Nonconvex optimization by fast simulated annealing." *Proceedings of the IEEE* 75.11 (1987): 1538-1540.