# Hyper-trace debugging for performance and security*

SAEID TIZPAZ-NIARI, University of Colorado Boulder
PAVOL ČERNÝ, University of Colorado Boulder
ASHUTOSH TRIVEDI, University of Colorado Boulder

Running time is a key concern in the performance and security analysis of programs. Performance debugging [4] and spectrum-based fault localization [23] are prevalent techniques to guarantee the performance of large systems. In the security context, the analysis of running times is necessary to guarantee availability [8, 10] and confidentiality [5, 7, 15] of programs.

Asymptotic time-complexity characterizations $O(f(n))$ are well-established metrics to express the worst-case running time in terms of the input size. Despite the importance of characterizing execution times with $O(f(n))$, there are often different modes in the execution times in terms of the input size. For example, the execution times of Apache FOP in Figure 1 (a) have the worst-case complexity $O(n^2)$, but there is another mode of execution time that is $O(1)$. This is a performance issue reported by a user in Apache bug forum (https://bz.apache.org/bugzilla/show_bug.cgi?id=51465). The question that an analyzer faces is "Do the differences in the execution times (the green and red patterns in Figure 1) manifest a performance bug?" If the answer is 'No', then the analyzer concludes that the differences are intrinsic to the application.

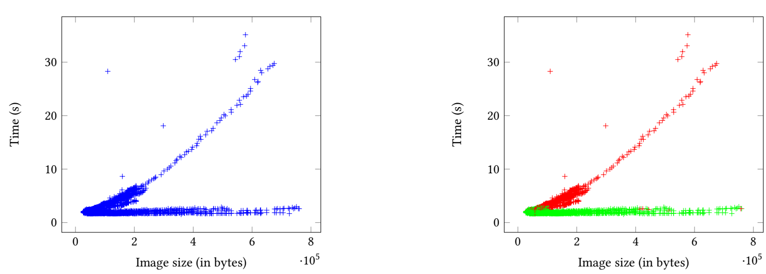


Fig. 1. (a) Apache FOP Execution Times versus input sizes. (b) Apache FOP Execution Times are clustered.

Execution times of programs depend on the programs' internals (i.e., control-flow graph) and their environments such as operating systems and hardware features. Differences in the execution times are the results of differences in the paths taken in the control-flow graph and the environments. Under a fixed environment, the variations in overall execution times of programs are explainable with some properties of program internals over its control-flow graph such as basic-block calls. In this work, we hypothesize that there are few groups of control-flow paths with distinguishing execution times, while there are many paths inside each group.

We define an input *trace* as a sequence of basic-blocks taken in a control-flow graph path. A *hyper-trace* is a set of input traces that expose similar execution time behaviors. *Hyper-trace debugging* is a novel technique for identifying and explaining differences in overall execution times among a set of input traces. The discovery part, first, clusters the inputs based on their execution times into several groups, and then, labels the corresponding input traces with the cluster information. Thus, this step identifies a set of hyper-traces where each corresponds to one cluster. Given the

*Position Paper

Authors' addresses: Saeid Tizpaz-Niari, University of Colorado Boulder, saeid.tizpazniari@colorado.edu; Pavol Černý, University of Colorado Boulder, pavol.cerny@colorado.edu; Ashutosh Trivedi, University of Colorado Boulder, ashutosh.trivedi@colorado.edu.

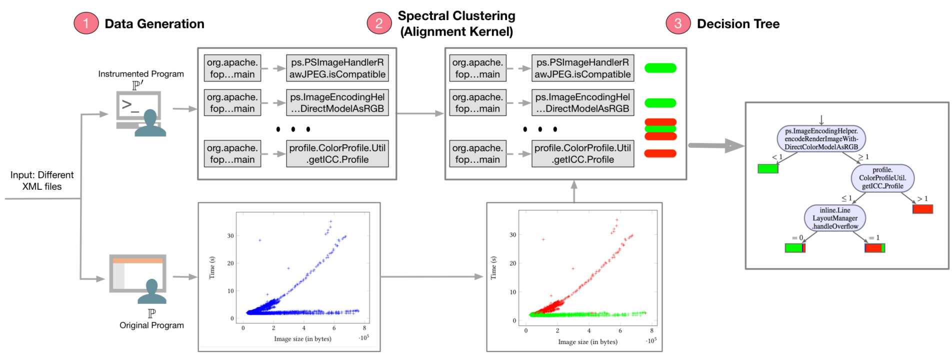


Fig. 2. Hyper-trace debugging for performance bugs in Apache FOP

set of hyper-traces, the explanation part produces a model that shows what properties of program internals are common inside a hyper-trace and what properties distinguish different hyper-traces.

The hyper-trace debugging is, while related, different than the profiling. In the profiling [4], the question is that given an input trace, how do the various parts of the program contribute to the overall execution time? The hyper-trace debugging, in contrast, looks for distinguishing properties of program internals that result in varying execution times among a set of inputs. The hyper-trace debugging is a complementary for testing. While testing tools such as fuzzers [1, 2, 16] can reveal performance bugs, the bugs are often difficult to analyze because they do not give knowledge about the cause of bugs in the program internals [13]. While the hyper-trace debugging can use the actual time measurements or abstractions such as the number of byte-code executed, the discriminant model uses the actual measurements to consider the environment that is not fixed in the real world.

**Debugging of Apache FOP**. We show how a hyper-trace debugging tool named DPDEBUGGER helps the user of Apache FOP to identify and explain the performance issue reported in the Apache bug database (https://bz.apache.org/bugzilla/show_bug.cgi?id=51465). The debugging procedure is shown in Figure 2. We collect a number of XML inputs that include PNG and JPEG images of various sizes. In the time domain, we apply spectral clustering [22] to discover classes of execution times as a function of input sizes. The clustering algorithm finds two classes of inputs with constant and non-linear functions as shown with green and red colors in bottom right of Figure 2. In the program internal domain, for the same inputs, we collect the input traces that are a sequence of method calls shown in the top left of Figure 2. Then, we label each input trace with the the cluster label from the time domain shown in the top right of Figure 2. The set of traces in green and red classes are the two hyper-traces. We use the decision tree algorithms to learn the discriminant model of the hyper-traces. The decision tree is shown in the right of Figure 2. For example, it pinpoints the method `getICCprofile` as one discriminant for red cluster. It is called for some PNG files that have a compressed color scheme that needs to be deflated. Now, the debugger can determine whether the performance differences are intrinsic to the application, or there are performance bugs.

**Research Statement.** The hyper-trace debugging technique with a data-driven approach of clustering and classification algorithms is a scalable, useful, and general-purpose tool for differential performance and timing side-channel debugging in software applications.

We dub the core problem in hyper-trace debugging *discriminant learning* problem. We define the problem and show the instantiations for 1) bug localizations, 2) differential performance debugging, and 3) timing side-channel analysis.

**Discriminant Learning Problem.** Given a set of executions, a *discriminant* is a map relating each execution time class to a formula over program internal features satisfied by the executions assigned to that class. Formally, let $\mathbf{X}$ be the set of (observable) input variables such as the size of inputs, $\mathbf{Z}$ be the set of auxiliary variables about program internals such as basic block calls, and $\mathbf{y}$ be the observable output variable such as execution time. An *execution trace* $\tau$ of the program is a tuple $\langle X, Z, y \rangle$. A trace discriminant $\Psi = (\mathcal{F}, \Phi)$ is tuple of a set of execution classes $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$—where each $f_j$ defines an execution time class over the output variable $y$—and a set of predicates $\Phi = \langle \phi_1, \phi_2, \ldots, \phi_k \rangle$ over the auxiliary variables $\mathbf{Z}$. A trace $\tau$ receives the execution class $f_j$ under trace discriminant $\Psi$ if $\tau \models \phi_j$, i.e. $\phi_j$ evaluates to `true` for the truth valuation of $Z \in \tau$.

**Discriminant Learning for bug localizations.** We study a discriminant technique for locating regions in the program that contribute to different observations. In the debugging community, this discriminant technique is an instance of spectrum-based fault localization [23] that is extended to hyper-traces. The output variable takes only $k$ distinct values and the values of input variables are the same for all program traces. The discriminant is to discover $k$ Boolean formulas over the auxiliary variables. An input trace belongs to the class of observation $j$ ($1 \leq j \leq k$), if $j$ is the smallest index such that the trace satisfies the predicates $\phi_j$. We refer to [20] for the complexity analysis of the discriminant learning with arbitrary boolean and monotone conjunctive formulas.

Due to the complexity of discriminant learning and noisy time measurements, we propose to use *maximum likelihood* discriminants over monotone conjunctive formulas. We propose two approaches to learn the discriminants: 1) integer linear programming, 2) decision tree classifiers. On a set of micro-benchmarks and case-studies, we show the decision tree learning algorithms such as CART [6] are efficient and scalable approaches to learn the discriminants [20].

**Discriminant Learning for differential performance debugging.** Now, we consider another version of discriminant learning problem where the execution-time classes are functions from the input variables to the output variable. This case identifies different *asymptotic performance classes* [12]. Given a trace and a discriminant model, we define the prediction error of a trace as the mean square errors between the prediction of the model and the true output. The goal is to learn the set of execution-time functions and corresponding Boolean formulas over program internals such that the prediction error is minimized with the smallest number of execution-time functions.

We propose *discriminant regression tree* (DRT) [19] approach to learn the discriminants. A discriminant regression tree is a binary tree structure whose nodes contain predicates over auxiliary variables $\mathbf{Z}$ and leaves contain a discrete probability distribution over the affine functions $\mathcal{F}$. Our approach is to first cluster traces to $k$ functions from input variables to output variable, and then assign different labels to various traces based on the *functional clusters* that they fall into. Next, we learn a decision tree in auxiliary variables with the leaves as clusters labels from the first step. We discuss two algorithms for functional clustering (see [19] for more details).

*K-linear Clustering.* Given the set of points, the K-linear algorithm uses an approach similar to K-means [14] to identify $k$ functional clusters with centroids $\langle f_1, f_2, \ldots, f_k \rangle$. Initially, the algorithm randomly picks $k$ pairs of points and fits $k$ linear regressions (initial centroids). Then, it assigns each point to the closet centroid and updates the centroids in each step until it converges.

*Alignment Kernel.* The alignment kernel is a notion of similarity between points where two points are close if the linear model fitted to them includes many other points within $\epsilon$ neighborhood. The spectral clustering [22] uses the kernel as the similarity notion to discover $k$ linear clusters.

**Discriminant Learning for functional side channels.** In the security context, the discriminant learning problem has two types of input variables: the secret inputs $\mathbf{X}_s$ and public inputs $\mathbf{X}_p$. In this setting, we define *functional observations*. A functional observation of a secret input $s$ shown with $\delta(s)$ is the execution-time function obtained by running the program on the entire set of public inputs $\mathbf{X}_p$. The discriminant model defines with the $k$ distinguishable classes of

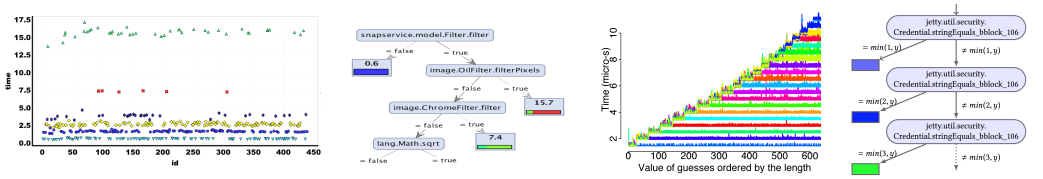


Fig. 3. (a) SnapBuddy Classes. (b) SnapBuddy Tree. (c) Jetty Observations. (d) Jetty Explanations.

observations $\mathcal{F} = \langle f_1, f_2, \ldots, f_k \rangle$ where each $f_j$ is the centroid of a set of functional observations that are indistinguishable from each other and their corresponding program internal formulas. Two functional observations $\delta(s)$ and $\delta(s')$ are indistinguishable and their secret values $s$ and $s'$ are in the same class if the distance $d$ between the functions is less than $\epsilon$. If all secret values are indistinguishable from each other, i.e., $k = 1$, the program is said to be functional non-interference under the timing observations. If there are more than one class ($|\mathcal{F}|>1$), the program is leaking some information about the secret inputs and the number of classes $k$ quantifies the amount of information leaks in accordance to min-entropy measure [18].

On a set of programs, we show that there are practical timing side channels with functional observations that might deem non-interference with the existing definitions [9]. Furthermore, we develop an approach based on the functional data clustering and decision tree classifiers for debugging functional timing side channels (see [21] for more details).

For a given secret value $s$, we use B-spline basis [17] to obtain the timing function $\delta(s)$ for the set of execution times over public inputs. For each secret value $s$, the valuations of each auxiliary feature is also a function in domain of public inputs. Given a set of timing functions and an arbitrary distance function $d$ with the tolerance $\varepsilon$, we apply non-parametric functional clustering [11] to identify $k$ functional observations. Using the auxiliary variables as (functional) features and the functional clusters as labels, CART algorithm learns the set of discriminants. The decision tree's nodes are program internal features and its leaves are a set of secret values inside a cluster.

**Examples.** First, we illustrate the usefulness of discriminant learning for bug localizations of SnapBuddy [20]. Then, we show the approach for the analysis of functional timing leaks in Jetty [21].

*Example 1.* SnapBuddy is a Java application with 3,071 methods, implementing a mock social network in which 439 users have public profiles with a photograph [3]. The inputs are the download requests for the profiles. The size of profiles are the same for all users. Figure 3a shows a scatter plot of the execution times for responding the requests. The execution times are clustered into 6 different groups using the $k$-means algorithm [14]. We see that for some users, the execution times were roughly 15 seconds, whereas for some others, they were roughly 7.5 seconds. We use the discriminant learning to discover what program internals are contributing to different clusters.

On the instrumented version of SnapBuddy, we obtain traces of inputs that are method calls. Then, we label each trace with corresponding cluster label and use the *CART* decision tree algorithm [6]. The decision tree model is shown in Figure 3b. The decision tree model explains that the filters applied by users on their profile images are discriminant properties for different clusters.

*Example 2 (Jetty).* We analyze the `util.security` package of Eclipse Jetty. The secret input is the password stored at the server and the public input is the guess for the secret. We use a combination of libFuzzer [2] and SlowFuzz [16], to generate the set of secret and public inputs. For each secret value, we use B-spline basis and learn the timing functions over the entire set of public inputs.

We consider $L_1$-norm distance between functions with $\varepsilon$=0.1 and apply a non-parametric functional data clustering. The clustering algorithm discovers 20 classes of functional observations as shown in Figure 3c. The existence of 20 clusters indicates the presence of functional side channels. Figure 3d shows the decision tree model. Using this model, the analyzer realizes that the execution of `stringEquals_bblock_106` is what distinguishes the clusters from each other.

## REFERENCES


[1] 2016. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

[2] 2016. libFuzzer - a library for coverage-guided fuzz testing (part of LLVM 3.9). http://llvm.org/docs/LibFuzzer.html

[3] 2016. Snapbuddy Application. https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges/Engagement_2/snapbuddy_1

[4] Thomas Ball and James R Larus. 1996. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 46–57.

[5] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S Păsăreanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *FSE'16*. ACM, 193–204.

[6] L. Breiman, J.H. Friedman, R.A. Olshen, and C.I. Stone. 1984. *Classification and regression trees*. Wadsworth: Belmont, CA.

[7] D. Brumley and D. Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.

[8] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 186–199.

[9] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *CCS, 2017*. 875–890.

[10] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks.. In *USENIX Security Symposium*. 29–44.

[11] Frédéric Ferraty and Philippe Vieu. 2006. *Nonparametric functional data analysis: theory and practice*. Springer Science & Business Media.

[12] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *FSE'07*. ACM, 395–404.

[13] Matt Hillman. 2013. 15 minute guide to fuzzing. https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing

[14] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis & Machine Intelligence* 7 (2002), 881–892.

[15] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[16] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2155–2168.

[17] James Ramsay, Giles Hooker, and Spencer Graves. 2009. *Functional data analysis with R and MATLAB*. Springer Science & Business Media.

[18] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 288–302.

[19] Saeid Tizpaz-Niari, Pavol Černý, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2018. Differential Performance Debugging with Discriminant Regression Trees. In *32nd AAAI Conference on Artificial Intelligence (AAAI)*. 2468–2475.

[20] Saeid Tizpaz-Niari, Pavol Černỳ, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Discriminating Traces with Time. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 21–37.

[21] Saeid Tizpaz-Niari, Pavol Cerny, and Ashutosh Trivedi. 2018. Data-Driven Debugging for Functional Side Channels. arXiv:arXiv:1808.10502

[22] Ulrike Von Luxburg. 2007. A tutorial on spectral clustering. *Statistics and computing* 17, 4 (2007), 395–416.

[23] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.