Safer Program Behavior Sharing through Trace Wringing

Deeksha Dangwal
University of California, Santa Barbara
deeksha@cs.ucsb.edu

Weilong Cui
University of California, Santa Barbara
cuiwl@cs.ucsb.edu

Joseph McMahan
University of California, Santa Barbara
jemcmahan@cs.ucsb.edu

Timothy Sherwood
University of California, Santa Barbara
sherwood@cs.ucsb.edu

Abstract

When working towards application-tuned systems, developers often find themselves caught between the need to share information (so that partners can make intelligent design choices) and the need to hide information (to protect proprietary methods or sensitive data). One place where this problem comes to a head is in the release of program traces, for example a memory address trace. A trace taken from a production server might expose details about who the users are or what they are doing, or it might even expose details of the actual computation itself (e.g. through a side channel). Engineers are often asked to make, by hand, "analogs" of their codes that would be free from such sensitive data or, may even try to describe behaviors at a high level with words. Both of these approaches lead to missed opportunities, confusion, and frustration. We propose a new problem for study, trace-wringing, that seeks to remove as much information from the trace as possible while still maintaining key characteristics of the original. We formalize this problem and show that, for a specific instance around memory traces, as little as a few thousand bits need to be shared. We demonstrate experimentally that the trace-wrung proxies behave similarly in the context of cache simulation but with bounded leakage, and examine the sensitivity of wrung traces to a class of attacks on AES encryption.

CCS Concepts • **Security and privacy → Information flow control**.

Keywords Privacy of traces, Synthetic trace generation, Trace compression

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304074

ACM Reference Format:
Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. 2019. Safer Program Behavior Sharing through Trace Wringing. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304074

1 Introduction

A quantitative approach to optimizing computer systems requires a good understanding of the way applications exercise a machine; real program traces taken from production code, in production environments lead to the clearest understanding. Unfortunately, even the simplest program traces, such as memory access patterns, have the potential to leak arbitrary information about the system. For example, a trace can capture the memory access behavior of a critical cryptographic function (which is known to be a function of the secret key [40]), a set of lookups corresponding to the parsing of a social security number, or even detailed system configuration parameters that are considered a trade secret. While the sharing of these traces between technology partners can lead to more robust and high performance systems, it can also leak highly sensitive information, and expose user data to security vulnerabilities.

It has been shown [47, 51] that safe ad-hoc anonymization is difficult to achieve. Given the cleverness of attackers working to undo well-intentioned, but ultimately insufficient, anonymization techniques [36], many have simply decided to cease making traces available altogether. Today when such traces are needed, programmers may be asked to "obfuscate" the key algorithm behaviors to hide sensitive data or provide "models" of the system which *approximate* the same behavior but omit sensitive parts. Hand-built "models" of the system are both tedious to code and of limited predictive power. Since there is no well-defined and well-trusted approach to this problem, developers are often forced to resort to rough human-language descriptions of the behavior of programs (e.g. "it is 80% pointer-chasing"). This leads to missed opportunities, frustrated optimization, and the design process ultimately suffers. Ideally, engineers would

access methods to eliminate any sensitive information from the traces while still capturing the program behavior and its interaction with the underlying hardware. However, the extent to which "sensitive" data influences program behavior is rarely understood by a single party, and even harder to argue is that it is completely absent from a trace.

We present a new formulation of this problem where one knows *a priori* exactly how much information a trace is giving away in the worst case. The basic idea is to take a trace and squeeze it through as small a "hole" as possible to extract as much information as possible out of the trace without completely compromising the usefulness of the trace. Like *wringing* all of the water from a sponge, in the ideal case only the *structure* of the trace (the dry sponge) remains and all potentially sensitive data has been eliminated. While we have no mechanism of quantifying the amount of sensitive data that remains, we do have a way to say how much *total* information is provided, which yields a useful upper bound. In other words, while we cannot say for certain how much water remains in the sponge, we know that the amount of water has to be strictly less than the total volume we squeezed the sponge into. We observe that when compression is taken to this extreme and lossy form, it connects to security in this unexpected way. However, as is often the case in computer architecture, an important tradeoff remains between information leaked and degree to which the trace accurately captures the behavior across a suitable domain of possible options.

We formalize this new approach specifically in the context of memory address traces, as they are well studied and we have many prior techniques to build from. To explore the tradeoff exposed by this problem, we examine a new approach of performing guided memory trace synthesis building on ideas from signal processing. By projecting the address space onto a wrapped 2D image, we are able to decompose memory behavior into an orthogonal set of features that can then be replayed to reproduce the same "visible" patterns as the traces under examination. Specifically, we use a Hough-transformed version of the trace to find both constant and strided access patterns; Hough features are also used to concisely summarize the trace behaviors. Our contributions:

1. We introduce trace wringing, a new paradigm of anonymity and privacy in the context of traces where compression and modeling provide a way to release information with easily verifiable bounds on leakage.
2. We demonstrate a pipeline instantiating this idea in the context of address traces and show how signal processing techniques can be used to squeeze information out of traces while maintaining program behavior.
3. We verify through cache-simulation results that trace-wringing can be achieved as a proof-of-concept. While the resulting systems may still give away thousands or tens of thousands of bits, it opens the door to further optimization and refinement.
4. We compare our approach with prior work in address trace compression and synthetic trace generation. We are able to construct proxy traces using as few as tens of thousands of bits which is orders of magnitude fewer than compressed traces and the profile used in synthetic trace generation.
5. As a first evaluation of security beyond just bit leakage, we show that a class of existing AES attacks fails to find useful information in the traces processed in this way, which illustrates the utility of such an approach.

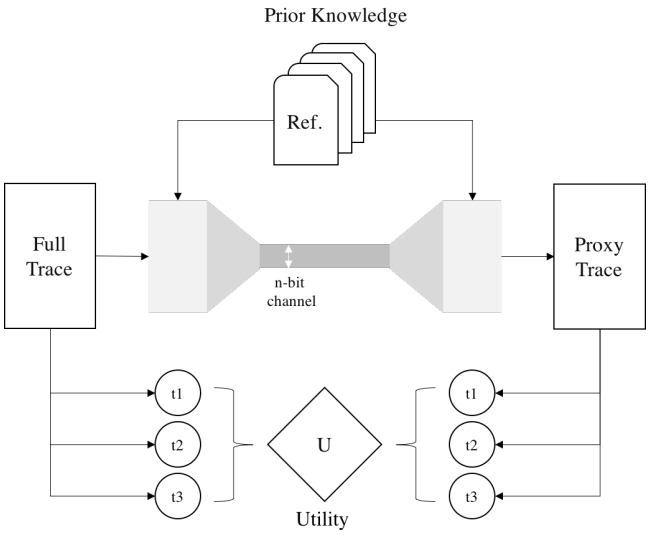

Figure 1. Forcing a trace through a channel with a capacity of only a few bits bounds the amount of sensitive data shared. While any public information such as prior non-private traces can be used in the creation of the code, the trace to be coded must not be known to the receiver. The objective then is to minimize the number of bits shared while maximizing the utility of the proxy trace. Here, we measure the utility in terms of whether or not certain tests *t1, t2,* and *t3* are passed by the proxy test and/or how close to the original tests results they get.

The rest of the paper is laid out as follows. First, we present the new problem of "wringing" a trace more completely. In Section 3, we compare and contrast this problem to its related work on prediction, compression, and other classic trace analysis approaches. Section 4 describes our approach of using signal processing techniques for trace wringing. In Section 5, we describe our experimental setup, followed by an evaluation where we compare cache-simulation results. We summarize and conclude in Section 6.

2 Wringing a trace

A program trace can contain a tremendous amount of information about the system under evaluation. For example, memory accesses give away the data (e.g. secret keys) used in calculating the addresses, simultaneous accesses to different data storage areas can give away important relationships (e.g. between an individual's access rights and fields of a data structure they are accessing), and so on. But, as we know, such traces are invaluable for performance evaluation because they demonstrate the way the system actually behaves in the face of the workloads it must actually handle.

While the behaviors are important at a high level, rarely are the specific elements of the trace critical. Rather it is the relationship between those elements and the proportions that they appear in the trace that is often the key. This is of course not a new insight, and many people have attempted to capture these behaviors with microbenchmarks [28] and other trace synthesis schemes in the past [53]. What we claim as new is the idea that we can formalize these schemes in such a way that it *bounds* the amount of information leaked about a system being traced.

The argument is simple: if we only share n bits about a specific trace then we cannot leak more than n bits about that trace. In practice, this means that if we share only a few tens of thousands of bits of information about the trace, then nothing beyond those bits has been leaked. While it is not a perfect solution (some information might be lost), it says something useful about the maximum amount of information that can be leaked. For example, it should be impossible to recover an extensive list of social security numbers, sensitive health information, or even an entire set of secret keys from such a trace. To maximize security one wants to give away as little data as possible about the trace. However, to maximize utility the opposite is true. Here is a new question for computer architects – how little can one give away from the trace while still being useful?

At first one might consider this to be exactly the problem of compression, and there definitely is a resemblance. Most compression schemes seek to perfectly replay a given input sequence by exploiting the fact that their inputs are far from completely random [6]. By understanding those common structures, for example the tendency for repeating patterns to occur [18], a more concise representation exploiting these structures is possible. Most modern compression algorithms start from a relatively blank slate and train a predictor of some form on the input as they process it. The duality between compression and prediction is pointed out by Chen et al. [10], who note that when you predict a value with high accuracy you can compress by storing an encoding that "the predictor is correct n times in a row" most of the time. Lossy compression is then a natural extension of this idea where the predictor is "close enough n times in a row".

However, even lossy compression schemes typically seek to minimize the error between the original trace values and the compressed trace values [35]. Here we have a problem that is different in two important aspects. First, while we want to keep the behavior of the trace to our tests the same, we may not care that the actual addresses themselves are similar. Second, we should be able to prime our scheme with data from other traces that do not contain a secret that we care about. In this way, we can think about this problem as attempting to decompose a trace into two aspects: a trace's "structure", and a trace's "data". The trace structure is what defines the hierarchy of patterns inherent to the trace that are useful for making statements about performance, while the trace data contains the specific set of addresses that makes the trace complete. The structure is all we really care to transmit and, when separated from the data, may be incredibly compact. The question then becomes, *how compact for how useful?*

Answering this question requires an analysis across two metrics: information and utility, as described in Figure 1. Information is surprisingly easy to quantify; it is the number of bits from the secret trace that need to be transmitted. Note that any number of bits about other traces or training data can be shared freely and even hard-coded into the receiver. Our approach is to describe traces as a probabilistic grammar of generators coupled with very high level accounting of behavior over time and account for bits in both the structure and parameters of this scheme. Quantifying utility is harder and more use-case specific. We define a distance function between cache miss-rates of trace vectors as one such function, but understand there are many other metrics one might use [2, 43, 53].

While this problem is generalizable, we are considering address traces for this initial class of experiments. While many other classes of traces might benefit, address traces are some of the most well studied and understood, and provide the most stable foundation for this new work to be developed upon and evaluated.

3 Related work

In this paper, we start with a security parameter (the number of bits we tolerate giving away) and analyze a program's behavior by studying its address trace to eliminate information that is not essential to describe its behavior down to that security parameter. At the heart of it, we want to accurately characterize a program's trace, and preserve only the bare minimum information, so as to not leak it unintentionally. This new problem can then leverage much of the related problems in the fields of trace compression, statistical program profiling, synthetic trace and benchmark generation, and data privacy and anonymity. In the rest of this section, we will compare and contrast our work with the large body of work that precedes it.

3.1 Trace compression and approximation

Trace compression is well studied. TCgen [5] has a compression ratio as high as 77, 000 for certain benchmarks. Lossless algorithms exploit sequentiality and spatiality, value prediction [4, 6, 7], perform loop detection and reduction [18], convert absolute values to offsets [27], and use clustering to improve compression [24]. ATC [35], a compression tool for cache-filtered addresses, is capable of both lossless (using bytesort) and lossy compression (using sorted byte-histograms).

Compressed compact representations are used to understand and predict program behavior. Larus's work on whole program paths [30] introduces a method to determine a program's dynamic control flow, using the SEQUITUR [37] compression algorithm. Chilimbi presents a similar scheme to effectively represent a program's dynamic data reference behavior [11], also using SEQUITUR. Trace Approximation [22] generates compact summaries of memory accesses of parallel applications to achieve trace reduction.

3.2 Characterizing program behavior

Eeckhout et al., have described a method to obtain detailed statistical profiles within program traces [17] with the combination of microarchitecture-dependent and -independent profiling tools. Their syntactically correct, and representative synthetic traces can be simulated on existing simulation tools. Machine learning algorithms are to understand large scale program behavior by clustering basic block vectors to find the representative sections of a program [45].

Chen et al., have shown that hardware event profiles for feedback-directed optimizations, can be improved by using machine learning and statistical techniques[9]. Oskin et al. collect statistics from actual program simulation to generate a synthetic benchmark [39] that is faster to run. While statistical methods are useful in modeling behaviors of programs, they do not consider the amount of information they inadvertently leak. It is worth revisiting these works in the context of how much total information they leak versus how useful they are across a range of optimizations. We leave unifying these approaches in the context of wringing as future work.

3.3 Synthetic trace generation

Synthetic trace generation has been a classic solution to characterize performance and effectiveness of novel designs (when workloads do not exist) [49]. To ensure that the synthetic traces behave as expected, Thiebaut et al. adhere to a hyperbolic probability law [42, 49]. Other methods on artificial workload generation have been described [19] and reviewed [20]. PSnAP [38] separates the program structure from the memory access pattern in two phases: capture, when PSnAP generates a profile using PMaCInst [50], and replay, when it produces a synthetic trace based on the captured profile.

For HPC applications, Weinberg et al. determine memory signatures and mimic them to generate synthetic traces [54]. They maintain the cache miss rates of the applications under test with Chameleon [53], a memory locality analysis tool suite. The tool produces a small seed, which is replicated to construct an arbitrarily long trace. BenchMaker [28] is a parameterizable and scalable synthetic benchmark generator, which can create customized workloads given some (forty) microarchitecture-independent program characteristics.

Unlike the previously discussed papers, BenchMaker creates *benchmarks* which can then be run on real-hardware (or simulators) in order to better explore the application space. Van Ertvelde et al. go further and propose code mutation [52] for generating benchmarks that hide functional semantics of proprietary programs. They do this at the binary level of chosen benchmarks rather than on traces.

3.4 Preserving data privacy

Differential privacy [16] protects anonymity by adding some amount of carefully calibrated noise to the sensitive data sets so as to maintain the main properties under study. Access to the system is metered out carefully to ensure privacy is maintained while being as true to the original distribution as possible. It has been pointed out recently [23], that differential privacy may introduce an unacceptable amount of error. *Being able to add noise to address traces in this fashion may not result in similar or expected program characteristics.*

Plausible deniability [3] presents a formal framework to generate synthetic data records efficiently while guaranteeing privacy. Their data synthesizer is based on a probabilistic model; it captures the joint distribution of attributes collected from the real dataset. Their target applications include machine learning and dataset analyses. Other formalizations of privacy are an active area of exploration with k-anonymity [48], i-diversity [33], t-closeness [31], and many others.

Traces are inherently time-series data sets. They map less clearly onto these models where a set of queries are often asked and answered by someone with the full data set. Unifying trace analysis and these models of privacy appears to be an open problem and our work stands out from the ones described here both by its intent and simplicity. We provide an up-front security parameter, the total amount of bits to be leaked, and we squeeze our traces to that level. This approach provides a useful point of comparison as more advanced techniques linked directly to more specific security models are developed and evaluated. Drawing inspiration from information theory, we also try to find an upper-bound on the information leaked from the system by trying to quantify the number of bits of information given away by our method while trying to minimize it.

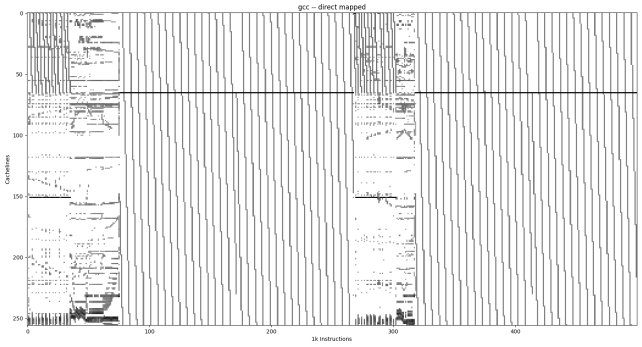

Figure 2. The modulo-memory access heatmap for gcc. The heatmap is an $N \times M$ sized graph, where N is some high power of 2 and M is the number of 10000 instruction windows in the trace. These *modulo-memory access heatmaps* illustrate patterns that exist within program executions, and give us a visual sense of memory access activity. When mapping longer traces, for example, we see phases (as in 4), but we also observe local patterns within these phases as shown here.

Another related field is quantitative information flow analysis; similar to differential privacy it proposes numeric measurements that pertain to privacy. Some examples of its applications are in producing better bug reports which maintain user privacy [8] and measuring source-location information leakage in wireless sensor networks [32] among many others. McCamant et al., present a method to determine how much information real programs leak [34] using a practical implementation of quantitative information flow which uses dynamic analysis.

4 A signal processing approach to wringing

Traces expose the inner workings of a program, its interaction with the runtime, and the underlying hardware architecture. As such, even the simplest memory traces prove to be a complex concoction of patterns generated by these underlying factors. For example, in a memory address trace, accesses to many different types of objects across both stack and heap are all interleaved to create the whole. Our goal of capturing the *structure* of these traces first requires that we identify, describe, and quantify the patterns that we care most about. While understanding the underlying cause of these patterns requires detailed knowledge of the program, quantifying the magnitude of these patterns can be done on the traces alone. In fact, it is observed that even complicated programs exhibit memory access patterns that can be decomposed into simpler ones.

To get a visual sense for the structure of such traces, we project the address trace onto a fixed-size modulo-mapping of the memory space. This *heatmap* is a graphical representation of the memory access behavior over time. Figure 2 shows such a heatmap for gcc where instruction count (time) runs along the x-axis and the address runs along the y-axis. If we were to plot this for the *entire* memory it would clearly be too large for such a graph (the distance between the stack and heap would dwarf any local behavior), so we instead plot the address modulo a large power of two. We call that the "wrapped address". This plot of the wrapped address over time (in terms of instructions) has the advantage of mapping addresses onto a more manageable space, but at the same time keeps the spatial-temporal structures that would actually impact a real cache. The *darkness* of each pixel is a function of the total number of memory accesses that happen to that wrapped address during a window of instructions.

Interesting and intuitive patterns emerge after looking over this graph. The flat horizontal lines in the graph are patterns of repeating access to a set of addresses. These are high temporal locality behaviors. Sharp diagonal lines, on the other hand, are regions of high spatial locality as addresses are accessed one after the other in succession. If we can concisely capture the *character* of these behaviors, without transmitting the addresses themselves, we can minimize the amount of information leaked. Describing an efficient method for extracting these patterns is exactly the goal of this section.

Figure 3 gives a high-level overview of the pipeline we propose to first wring and then expand a trace. There are two essential subsystems in our pipeline; one for extracting structural information about the trace from our heatmaps, i.e., for trace-wringing, and the other for rebuilding a proxy trace with the same structural information. At one end, as seen in Figure 1, with the help of some prior reference knowledge about traces, a full trace is decomposed into its describing parameters. These parameters are the ones being communicated via a constrained channel to the generator subsystem, which then uses the same prior reference knowledge and the descriptive parameters to generate a proxy trace. In our pipeline, prior reference is used for optimization of encoding (generation of heatmaps, detection of phases and line segments within them, and creation of "information packets"), decoding (proxy trace generation from shared "information packets"), and the selection of Hough parameters. The generated proxy trace's utility is measured by testing its properties against that of the original full trace.

The modulo-memory heatmaps exhibit hierarchical organization. Globally, there exists a recurrence of similar patterns in the order of a few tens of thousand instructions, i.e., the presence of program phases, and within them, we observe patterns that we associate with the more local memory access activity. In order to find some representative of the higher echelons of this hierarchy, we employ k-means clustering to detect the program phases [45].

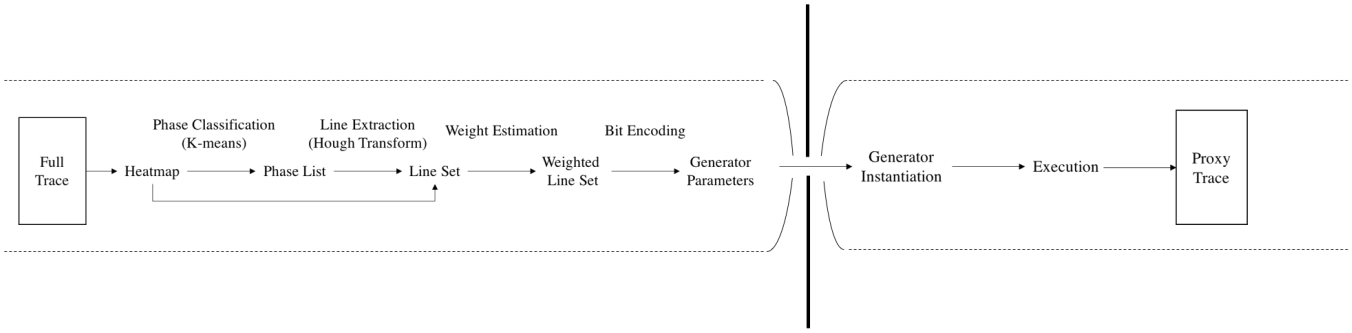

Figure 3. Pipeline for our signal processing approach to trace-wringing for proxy trace generation. The problem of sharing information can be described with two subsystems; at the trace-wringing end, we find parameters that will accurately generate the trace at the generator subsystem end. The goal is to minimize the size of the packets being sent between the two subsystems, while still maintaining integrity of the data transmitted.

4.1 Phase detection

While Figure 2 is not the full execution of gcc, we note the presence of a set of program phases. The first observation we make is that if we wish to capture the character of these traces, we need to extract higher level shifts in behavior over time. If one can group together alike behaviors (for example, the middle and end of Figure 2) we can then select only a *single representative* for each such behavior. Fortunately this is almost exactly the problem of phase detection [14, 44, 46]. To find the phases, and select a representative, we pose this as a clustering problem (similar to prior work). We break the execution up into a set of "chunks" by instructions executed. The columns of the chunks are then summed together to form a vector. Each vector thus has a length equal to the number N of wrapped line addresses. We can think of each of these vectors then as a point in N dimensional space. Finding groups of similar points (our memory vectors) is then exactly the clustering problem. Here we can simply apply the k-means algorithm [25] with k equal to the number of phases we wish to represent in the trace. The k-means algorithm represents clusters by a set of k cluster centroids which it then iteratively optimizes. Each iteration alternates between assigning each point in the space to exactly one centroid, and updates centroid position to be in the "middle" of the new set. After k-means, we take each cluster and select one that is the longest to be the representative cluster.

Figure 4 shows the result of running the phase detector on the memory address trace for gcc. Each of the 3 colors labels the trace above it with a unique phase identifier. The technique does a good job of lining up with the repeating structures.

Now, with these phases marked, rather than encoding the full trace monolithically, we can encode just the k representative clusters independently with $log_2 k$ bits. The list of the phase identifiers can then become part of the information shared. As can be seen in Figure 4, there is a great deal of temporal locality in the phases and can be trivially compressed by another order of magnitude with run-length encoding.

Given that we now have a set of representative chunks of execution, we need to efficiently summarize the features that exist within each chunk. If we look back to Figure 2, we can see that many of the patterns in the heatmap can, in fact, be reduced mostly to a set of lines.

4.2 Decomposing with Hough transforms

Concisely summarizing all of the complex patterns of the trace all at once can be overwhelming. However, if we can break the pattern down into a set of simpler behaviors, we can then tackle them one by one. Given that both strong temporal and spatial locality features show up as lines, decomposition into a set of line segments is a natural place to start. However, decomposing the address trace features in the space of $wrapped\_addresses \times instruction\_count$ directly is not easy. Luckily, we can draw upon established methods in image processing to transform our heatmaps into a space where such extractions are achievable.

The Hough transform [15] is a popular computer vision procedure used to detect patterns in images. The technique is used to find the locations and orientations of certain geometric primitives in the given space. Hough transforms, being resilient to noisy images, makes for an ideal feature extraction candidate for our problem. Geometric primitives such as lines, ellipses, and circles are supported by Hough transforms, but we find use only for the simplest Hough transform: the Hough-line Transform.

While standard regression methods are useful fitting a slope-intercept form of $y = mx + b$ to a set of points, finding *sets* of rotated lines from an image is hard in the Cartesian coordinate system. The Hough-line transform employs the polar coordinate form and describes lines by their distance from the origin r and the angle formed between the origin and the closest point on the line θ: $r = x \cos\theta + y \sin\theta$.

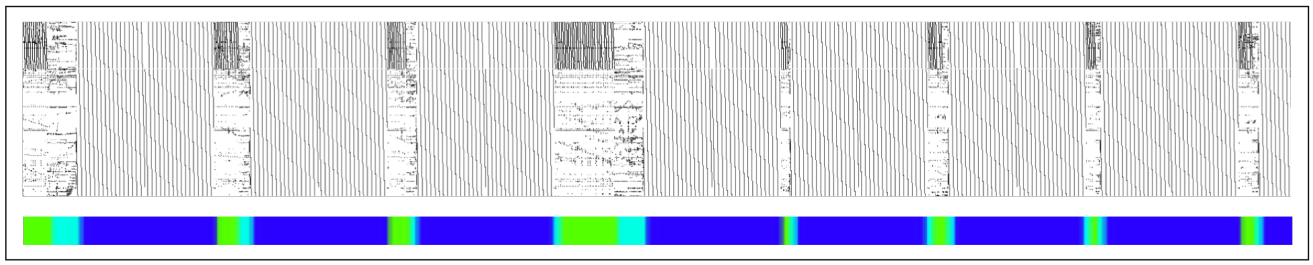

Figure 4. Phases visible in the trace generated by gcc after k-means clustering. Each of the 3 colors in the bottom marks a unique phase in the trace. Note, importantly, that phases reoccur over time.

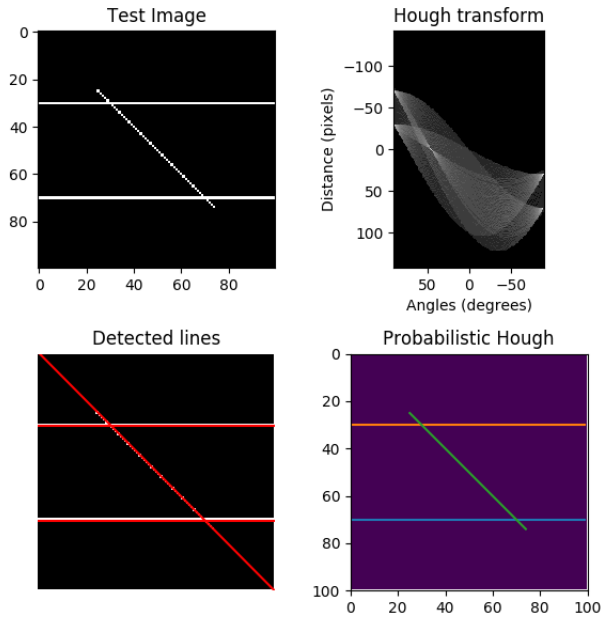

Figure 5. We capture information about lines we observe in trace heatmaps using the Hough Transform. Here, we demonstrate its working. The points on the test image are surveyed for parameters in the polar coordinate space described as the Hough Transform. The intersections describe the parameters of the detected lines. The final figure shows the Probabilistic Hough Lines, the more robust and efficient algorithm. For our heatmaps, we use the Probabilistic Hough Line algorithm.

Now, we have two separate coordinate systems in which we can find the best fit line; the image space, and the $<r, \theta>$ parameter space. For every point in the image space, the Hough transform considers every possible rotation of lines passing through that point. Iterating through the different possible values of r and θ in the Hough space, the algorithm forms a sinusoidal curve for each point in the image space. Each point in the $<r, \theta>$ space corresponds back to one possible straight line in the image space. This point-to-curve transformation (where every point in the image space is a curve in $<r, \theta>$ space) is the Hough-line transform. We do this for all the points, and the most coincident points (where the most sine curves intersect) in the $<r, \theta>$ space is the choice of parameters for a line in the image space. Specifically, what makes the Hough transform robust is how the parameter space is set up: it is divided into a mesh of finite intervals or accumulator cells. As the algorithm proceeds from point-to-point in the (x, y) (image) space, the accumulators in the discretized $<r, \theta>$ space are incremented.

For our instance, we use the progressive probabilistic Hough transform [21], a rendition of the Hough transform algorithm that only performs voting on a subset of the input points. These input points are chosen based on certain features of the expected result, such as a threshold of "darkness", the length of the expected line, interpolation strategies, and the angle of the line. By interleaving the voting process with line detection, this algorithm finds the most prevalent features first, while also minimizing the computational load.

The progressive probabilistic Hough transform returns a set of lines, with each line's (x, y) coordinates in the modulo-memory heatmap space. We also introduce a variable, "weight", for each line, which is a measure of darkness of the line.

The list of phase identifiers (the result of clustering), the two (x, y) coordinates of each line detected by the Hough transformation, and the line's weight per representative phase, give us the amount of share-able information.

4.3 Proxy trace generation

Using phase detection and Hough-line transformation, we end up with a set of Hough lines for each representative phase. Each phase is also assigned a label indicating to which cluster it belongs to, i.e., which representative phase "represents" it. Since the structural information of each phase is encoded in the the Hough lines, we can generate an "address tracelet" for each phase using the representative's Hough lines.

Phases from the same cluster may occur intermittently and in different lengths. For all phases in the same cluster,

we generate patterns continuously in a rotating fashion regardless of the length. For example, if phases x_1 and x_2 are both represented by representative phase r_1 (suppose x_1 occurs before x_2 and there's no other phases represented by r_1 in between), we then generate a trace for x_2 following the partial patterns we generate for x_1 and wrap over if the total length grows beyond r_1, i.e., the starting time step t when generating addresses for x_2 will follow the end time step $t-1$ when we generate for x_1 and wraps over when t becomes larger than the end time stamp in r_1.

Within each phase, we generate addresses by alternatively picking addresses from the subset of lines that cover each point in time (each time step t in the projected address space corresponds to N addresses, in which N is determined by the window size when the heatmap is generated at first place). If there are no lines covering the current time step t, we generate addresses for t from a uniformly distributed noise function as there is no clear pattern observed by the Hough transformation and we mimic a random access behavior in this way.

Upon picking a Hough line at time t, we generate an address "segment" from that line based on a fixed segment length, which captures locality at a small granularity. The segment length for each workload is hand-picked so that it best captures characteristics of the trace. Each address generated from the line is also shifted to the left by the cache block offset bits (6 bits for a typical 64B line size) since the purpose of wringing is to preserve the cache-level patterns.

After generating address tracelets for all the phases, we concatenate them together in the original order of the phase occurrences to form a complete proxy address trace. The proxy trace has the same length as the original trace but its memory footprint is limited to the wrapped address space.

5 Evaluation

To evaluate the effectiveness of the approach, we take a set of traces, wring them through our pipeline to a target number of bits, and evaluate the traces across a range of cache configurations with regards to miss rate. The details of the parameters and process follow below.

Starting with the full traces, we first convert them into heatmaps which are parameterized by the number of instructions from the trace to simulate, the window size, and the total size of the mapped space. If a map space is chosen to be too large, the line detection techniques will fail to pick up useful edges as there is too much white space for them to operate properly. If the map space is too small then the addresses will be truncated to such a degree that they will cease to be useful for evaluating miss rate. For our experiments, the x axis in the modulo-memory heatmap represents 10,000 instructions.

We use signal processing techniques here to collect important information about the heatmaps. We compute the

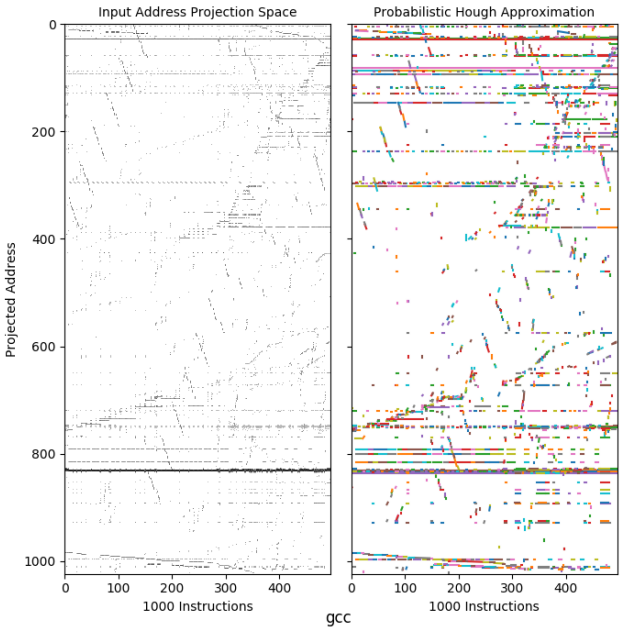

Figure 6. Producing probabilistic Hough lines on top of the heatmap of the SPEC2006 benchmark, gcc. The colors are used to indicate distinct lines produced by the decomposition.

Hough transforms, as described prior, to give us the value of the constants that describe the lines that the algorithm is able to "see" in the heatmaps. Specifically we must hand-tune the progressive probabilistic Hough transform input points (to reduce the search space of the algorithm) to find the lines in the midst of all the noise that these heatmaps inherently have. For our experiments, the parameter *threshold* ranged from [20,200], *line_length* ranged between [10,60], *line_gap* ranged between [1,50], and *theta* ranged between π and $\pi/2$. Specifically, the probabilistic Hough lines [41] are then generated and remapped back into the address space.

5.1 Measuring bits

While our main goal so far has been to extract and describe the structure of traces as correctly as possible, we must also maintain that not too much information is given away. The information that needs to be transmitted to the trace generator must contain both the global phase-identifier information, and the line coordinates and weights per representative phase.

$$Phase\_bits = \lceil log_2(\#\_phases) * len(phase\_seq) \rceil \quad (1)$$

To calculate the bits that are needed to produce the proxy trace for each workload, we dump all the labels from the clustering result as well as all the Hough lines detected, each of which is a 5 tuple of coordinates in the heatmap

space and a weight value. The phase information can be represented using *Phase_bits* (Eq. 1). We then apply a variety of compression techniques to compress the dumped files and estimate the bits of information by measuring the size of the compressed file. We push all of the information that is to be measured into a single file to ensure that no side information is accidentally shared between the two halves of the system. We discuss the breakdown effects of each compression technique in Section 5.4.

5.2 Trace selection

Rather than working on the traces in their entirety, for each workload, we evaluate from a large SimPoint [45] trace of the most representative region of 100M instructions, which results in a variable length of address traces from 30M to 70M accesses for different workloads. We use benchmark subsetting suggestions [29] to reduce the space of evaluation to a more manageable level, although our results are limited to 6 of the 9 suggested due to errors getting the benchmarks running. Results from all benchmarks run are considered and the optimal (in terms of bits leaked and accuracy of miss rate) points at two different levels of bit transmission budget are shown in Table 1. The time overhead for our pipeline is also presented in Table 1. Although it varies between different workloads, we expect this overhead to grow sub-linearly as the trace becomes longer for any single workload. The time overhead is linearly correlated with the number of distinctive phases in the trace and the number of phases tends to grow very slowly since phases often repeat themselves.

5.3 Measuring utility

As we concentrate on cache behavior as a target for initial evaluation we use cache miss rates pre-wringing and post-wringing to evaluate how useful the resulting trace is. The collected address traces are simulated with different cache configurations using DineroIV [26]. We use 6 cache configurations in our experiments: direct-mapped and 4-way associative combined with 3 different cache sizes (8k, 16k and 32k), and measure their miss rates.

From Table 1, we observe that as the bits of information leakage increase, the miss rate gets closer to the ground truth miss rate, which confirms that, with more information going through the wringing "hole", the proxy trace we reconstruct becomes more similar to the original trace in terms of structure. Some benchmarks such as *sjeng* and *hmmer* do not benefit much from the extra bits, in terms of closeness to the miss rate, as 10,000 or even fewer bits are *enough* to accurately capture their cache behavior, while others including *libquantum* perform much better due to the fact that they have a more complex structure which requires more bits to encode.

Figure 7 compares the proxy heatmap generated for gcc against the original. Our wrapped address space is of height 2048 (lines in the heatmap) and each "column" in the heatmap corresponds to 10,000 memory accesses. The figure illustrates that our approach is able to capture all but the subtlest patterns.

5.4 Comparison to existing compression and trace generation techniques

We are not aware of any prior methods that have attempted to bound the information leakage from generated traces. While our approach to bounding draws from trace compression and synthetic trace generation techniques, we stand out in at least the following ways: (a) we seek similar *behavior* in our generated traces, rather than similar addresses, (b) we allow unbounded priors from non-sensitive traces, (c) our traces are lossy specifically in a way that it maintains architectural utility, and (d) qualitatively, the target size of the final "compressed" trace is far smaller than normally considered. This last point, (d), is something that we can quantify experimentally.

Specifically, we compare our method against a state-of-the-art lossy compression and synthetic trace generation in Figure 8. "ATC" is an open-source implementation of the address trace compression framework [35], which supports lossy compression over cache traces. We run both off-the-shelf ATC, and a hand-tuned version that attempts to further minimize the trace size while still decompressing into useful traces. Although off-the-shelf ATC achieves good accuracy, it requires up to tens of millions of bits to represent the structure and data of the original trace in most cases. Even the hand-tuned version, which adjusts the similarity threshold and reduces the size of the unit of comparison, does not change the result significantly. This is orders of magnitude more than the number of bits transmitted in our trace-wringing framework (note the base 10 log scale). For synthetic trace generation, we use an open-source implementation of the Chameleon framework [54]. The profiles/characterization of traces are quite large even after h5 compression due to the fact that a histogram of address reuse is entirely captured in order to generate a similar-behaving synthetic trace. "FP+RLE+BZ2", our most aggressive post-wringing compression technique, significantly reduces the number of bits while maintaining good accuracy. This is not to say that these and related approaches could never be improved to be competitive on this new problem, but both out of the box and with some careful tuning, they do not appear to be currently.

5.5 Case study: AES attack

While it is impossible to say with certainty what could be leaked in the resulting bits, it is worthwhile to examine the technique practically in the context of a known attack. Specifically, we choose to examine the trace to see if it is possible to recover an AES key using known attacks. AES attacks based on cache sets have been well-studied [40]; we follow a similar process here.

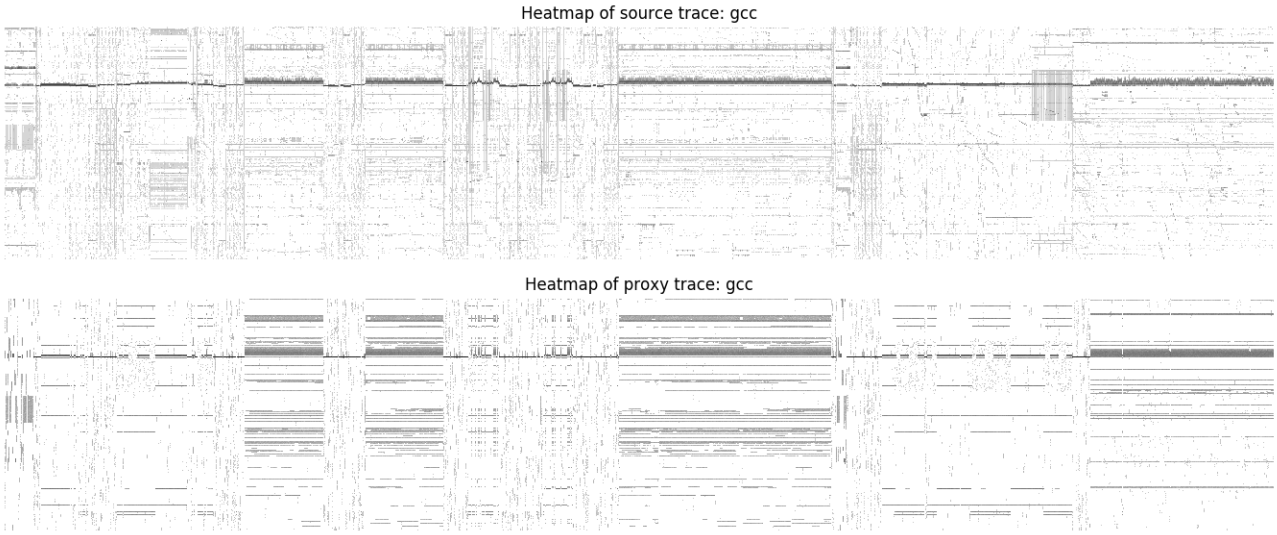

Figure 7. Heatmap for the original gcc trace and the trace-wrung proxy generated for gcc trace from the wrapped address space. Each pixel corresponds to one wrapped address at one time step. The darker the pixel, the more times that address is accessed during that time step.

The vulnerable portion of an AES trace lies in the accesses to the Rijndael substitution function (sbox). This is stored as a table in memory. In the first round of encryption, the offset into the table is the result of each byte of the key xor'd with each byte of the plaintext. When the attacker chooses or knows the plaintext, the offsets are of obvious importance — the ability to discover the table offsets directly leads to discovery of the secret key. Because the post-wringing trace consists of cache set indices, we limit the attack on the original trace to cache sets only as well for a fair comparison.

The attack model is as follows. Assume the attacker has chosen a uniformly random plaintext, and made N calls to an AES encryption, where each call has 16 bytes of the plaintext. The attacker can observe the resulting traces, either pre- or post-wringing. The attacker prepares a table of 256 "candidate" values for each byte of the key. Then, for each key byte, the attacker considers every address in the traces that could potentially fall within the sbox table. Each of these addresses corresponds to an sbox table offset, and, when xor'd with the appropriate plaintext byte, yield a candidate key byte. The corresponding entry of the candidate table is incremented by one. When finished, the key byte with the highest candidate score is used in the key guess.

The vast majority of addresses processed will not be sbox accesses; however, because the plaintext is chosen to be random, these will become uniform random noise. Only the first-round sbox accesses always come out to the same value when xor'd with the random plaintext: the correct key byte. With enough traces, the signal corresponding to the correct key will rise above the noise and be readily apparent. In our attack, looking at full addresses, it took only 13 encryptions to get all bits of the correct 16-byte key.

Since the post-wringing trace is a smaller space of bits, we are unable to attack full addresses. Instead, we attack the bits provided; this makes the attack very similar to the original cache attack [40]. Attacking the first round of AES cannot yield all the bits of each byte of the key, since the offset within a given cache set is unknown. Attacking subsequent rounds of AES can provide the rest of the bits, but requires that the first round attack is successful. Therefore, showing that the attacker is unable to succeed in attacking the first round is sufficient to demonstrate that the attack fails.

We perform this attack on a set of traces collected from runs of Tiny AES [1] with a random plaintext. We perform the same attack pre- and post-wringing. In the pre-wringing trace, we use only 12 bits of the address (the amount of information contained in the post-wringing trace), masking the lower three bits and the upper bits of the address. We note that this trace was wrung with 8-byte cache lines specifically to give advantage to the attacker and show the usefulness of the approach; increasing the cache line size only makes the attack more difficult. Pre-wringing, the attacker correctly guesses the upper five bits of all 16 key-bytes after 1,838 encryptions. This is the maximal information that can be learned in a first-round attack with 8-byte cache lines. Post-wringing, the attack guesses wrong for all 16 bytes of the key after 50,000 traces.

We performed an entropy calculation on the original traces based on the distribution of addresses at each time step across a number of traces. We see that ~160 addresses have more

Table 1. Best miss rates observed for the benchmarks with three different bit-budgets of information leakage and time overhead for trace-wringing followed by proxy trace generation. For each cache configuration 4 miss rates are reported. We report: ground truth miss rate from the original trace, best miss rate using *all* hough lines, best miss-rate with 100k bits, and best miss-rate with merely 10k bits. "-" means the most aggressive setting in our experiments requires more bits to construct the proxy traces.

Benchmark	Bit Budget	Cache Configs.						Time	
		8k,dm	8k,4w	16k,dm	16k,4w	32k,dm	32k,4w	Wringing	Decompression
gcc	Orig.	6.88%	3.91%	4.86%	2.79%	3.36%	2.11%	138.55s	123.37s
	Full	6.10%	3.98%	3.60%	1.27%	1.93%	0.48%		
	100k	4.82%	2.94%	2.81%	0.72%	1.40%	0.25%		
	10k	-	-	-	-	-	-		
sjeng	Orig.	12.3%	5.01%	6.45%	2.19%	4.24%	0.64%	94.42s	128.08s
	Full	12.85%	10.16%	8.22%	3.74%	4.26%	0.64%		
	100k	12.85%	10.16%	8.22%	3.74%	4.26%	0.64%		
	10k	11.89%	7.78%	1.13%	4.39%	0.25%	2.25%		
cactusADM	Orig.	8.29%	7.03%	5.44%	5.29%	2.09%	1.54%	209.94s	918.04s
	Full	9.35%	4.98%	5.21%	0.85%	2.08%	0.29%		
	100k	3.73%	0.49%	2.02%	0.14%	0.55%	0.12%		
	10k	-	-	-	-	-	-		
milc	Orig.	7.99%	7.09%	7.68%	7.03%	7.35%	6.94%	336.41s	31.36s
	Full	7.73%	7.19%	7.11%	6.66%	5.93%	5.69%		
	100k	7.51%	7.25%	6.75%	6.44%	5.46%	5.44%		
	10k	-	-	-	-	-	-		
hmmer	Orig.	27.8%	2.54%	26.8%	1.20%	17.0%	0.78%	151.79s	287.95s
	Full	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
	100k	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
	10k	23.6%	7.21%	20.53%	5.05%	10.31%	4.32%		
libquantum	Orig.	16.3%	16.2%	16.2%	16.2%	16.2%	16.2%	57.73s	21.89s
	Full	17.31%	17.27%	14.99%	14.90%	12.10%	11.90%		
	100k	17.31%	17.27%	14.99%	14.90%	12.10%	11.90%		
	10k	74.46%	74.44%	69.33%	69.31%	59.31%	59.32%		

than 5x the information content of the remaining addresses. These higher information-content addresses correspond to the sbox computations. Post wringing, all addresses have uniform information content, i.e., there is no set of addresses that is more influenced by the key than others.

Our wringing process was able to produce a new trace with comparable cache miss rates. We received 0.0% (new trace) against 0.9% (original trace) for the direct mapped cache and 0% (both new trace and original trace) on the 4-way associative caches while completely stopping our AES cache attack.

6 Conclusion

The conflict between the need to share information (to provide more optimal performance) and hide information (for privacy) is becoming increasingly fundamental in the computer system fields. While addresses are one such type of trace, one can certainly understand how related problems exist with storage traces, cache coherence traffic, energy usage, user interaction data, and certainly location data. Clever, yet complex, techniques have been developed to address certain anonymity problems in the past, yet the reality is that they are often dependent on specific assumptions such as a lack of prior information, statistical distributions governing the data, or that number of queries can be tightly bounded. While our wringing approach is very direct, that directness also comes with clarity as to what it does and does not do. It does not *guarantee* anything about how useful the resulting trace will really be for optimization. However, it *does* transform the problem of safe sharing into a *measurable* systems problem subject to the myriad tools we have at disposal for *common-case optimization*. Furthermore, it *does* provide a *strong and clear bound* on the amount of useful information given by the trace.

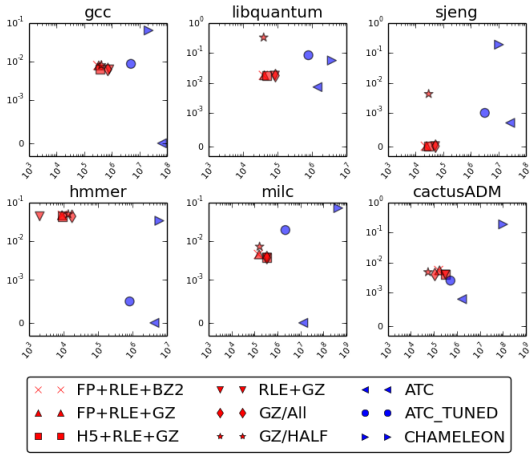

Figure 8. Breakdown of trace-wringing pipelines and comparison against state-of-the-art compression and synthetic trace generation techniques in the bit-error space. The x-axis represents *number of bits* transmitted, y-axis represents the *geometric-mean* of error in miss rate. Per workload, we mark the bit-error points for different techniques; being in the lower-left is better. A packet contains information about hough lines and labels. "FP" is fixed-point quantization on hough lines, "RLE" is run-length encoding on labels, "H5" is the HDF5 format compressed using h5py [12] for hough lines. We use a general purpose compressor on our packets, either Gzip,"GZ", or Bzip2, "BZ2". "GZ/ALL" and "GZ/HALF" indicate Gzip on unquantized packets of either all or highly-weighted half of the hough lines. "ATC" is the off-the-shelf lossy compression [35], "ATC_TUNED" is hand-tuned to minimize information transferred. "CHAMELEON' is from the open source implementation of Chameleon [53]

The technique we present here is a proof-of-concept and we make no claims that it captures anywhere near the true minimum leakage to utility tradeoff. There is much work left to be done to bring the number of bits shared compared to the accuracy lost down into a more appealing tradeoff. 10, 000 bits, let alone 100, 000 bits, is still a tremendous amount of information to leak and it is far from certain that it can never be used for anything malicious. From a security standpoint, we must do far better than that. Despite this gap, we feel that even these results are better than the other approaches, which fall to the extreme of either leaking almost no information with limited connection to reality or direct connection to observed behavior and completely unbounded information sharing. We establish this experimentally in Section 5 by comparing against existing approaches, which while designed for different purposes, do functionally provide a bit-reduced trace with diminished fidelity. The specific set of techniques we propose push the traces to much lower levels of leakage than these other past works can achieve with only slight losses in accuracy. This is perhaps not surprising as the levels of "compression" one needs to achieve to store a trace efficiently on disk are far less than that needed to have confidence there is little sensitive information retained.

Looking forward, with this new approach we can build on years of community experience dealing with address traces and encode common patterns in a general way. In many important applications, striding memory behavior is an important component and we believe we are the first to connect the address trace analysis problem with the Hough transform. The resulting analysis is surprisingly robust to noise and can capture general striding behavior. While this approach is effective for the memory problems we examined, there is no shortage of opportunity to build on the techniques we lay out to create more robust and higher quality trace wringing systems. Fully leveraging the best synthetic trace, trace compression, and statistical modeling techniques and understanding what they each bring to the problem is one next step. Bringing the full algorithmic power provided by the fact that *any* public trace data can be leveraged in the compression is also very promising. This opportunity is particular interesting as it sits outside of any past lossy compression or synthetic trace scheme's ability to exploit (i.e. minimizing total data transferred is different than minimizing sensitive data transferred). Further forward, we see a set of access behaviors (uniform random, stride, etc) that might form a set of "basis functions" which then are composed to describe a set of traces. Finding the *best* set of basis functions and how to *optimally* compose them to form good proxy traces can lead to many interesting follow-on works. It remains to be seen just how small of a footprint is achievable, but we believe there are orders of magnitude of improvement left to be had. Luckily, because the data to train such a wringing approach is generated completely by machine, this is an area where there is a great opportunity to gather a great deal of data to inform our models. The exploration of the hyper-parameter space of the wringing process can be automated using existing frameworks (e.g., [13]). In the end, this paper is a stepping stone to more general methods for trace sharing and we hope the clear metrics for success (e.g. share as few bits as possible) prompts further discussion and effort by the community.

Acknowledgments

The authors would like to thank Chandra Krintz, Lieven Eeckhout, and the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grants No. 1763699, 1740352, 1730309, 1717779, 1563935.

References

[1] 2014. Tiny AES in C. https://github.com/kokke/tiny-AES-c
[2] Erik Berg and Erik Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis*

of Systems and Software, 2004 IEEE International Symposium on-ISPASS. IEEE, 20–27.

[3] Vincent Bindschaedler, Reza Shokri, and Carl A Gunter. 2017. Plausible deniability for privacy-preserving data synthesis. *Proceedings of the VLDB Endowment* 10, 5 (2017), 481–492.

[4] Martin Burtscher. 2004. VPC3: A fast and effective trace-compression algorithm. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 32. ACM, 167–176.

[5] Martin Burtscher. 2006. TCgen 2.0: a tool to automatically generate lossless trace compressors. *ACM SIGARCH Computer Architecture News* 34, 3 (2006), 1–8.

[6] Martin Burtscher, Ilya Ganusov, Sandra J Jackson, Jian Ke, Paruj Ratanaworabhan, and Nana B Sam. 2005. The VPC trace-compression algorithms. *IEEE Trans. Comput.* 54, 11 (2005), 1329–1344.

[7] Martin Burtscher and Metha Jeeradit. 2003. Compressing extended program traces using value predictors. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. IEEE, 159–169.

[8] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. 2008. Better bug reporting with better privacy. *ACM SIGARCH Computer Architecture News* 36, 1 (2008), 319–328.

[9] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 42–52.

[10] I-Cheng K Chen, John T Coffey, and Trevor N Mudge. 1996. Analysis of branch prediction via data compression. *ACM SIGPLAN Notices* 31, 9 (1996), 128–137.

[11] Trishul M Chilimbi. 2001. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 191–202.

[12] Andrew Collette. 2013. *Python and HDF5: Unlocking Scientific Data*. " O'Reilly Media, Inc.".

[13] W. Cui, Y. Ding, D. Dangwal, A. Holmes, J. McMahan, A. Javadi-Abhari, G. Tzimpragos, F. Chong, and T. Sherwood. 2018. Charm: A Language for Closed-Form High-Level Architecture Modeling. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 152–165. https://doi.org/10.1109/ISCA.2018.00023

[14] Ashutosh S Dhodapkar and James E Smith. 2003. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 217.

[15] Richard O Duda and Peter E Hart. 1972. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM* 15, 1 (1972), 11–15.

[16] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*. Springer, 1–19.

[17] Lieven Eeckhout, Koen De Bosschere, and Henk Neefs. 2000. Performance analysis through synthetic trace generation. In *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*. IEEE, 1–6.

[18] EN Elnozahy. 1999. Address trace compression through loop detection and reduction. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 27. ACM, 214–215.

[19] Domenico Ferrari. 1981. A generative model of working set dynamics. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 10. ACM, 52–57.

[20] Domenico Ferrari. 1984. *On the foundations of artificial workload design*. Vol. 12. ACM.

[21] C Galamhos, Jose Matas, and Josef Kittler. 1999. Progressive probabilistic Hough transform for line detection. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, Vol. 1. IEEE, 554–560.

[22] Xiaofeng Gao, Allan Snavely, and Larry Carter. 2006. Path grammar guided trace compression and trace approximation. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*. IEEE, 57–68.

[23] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire.. In *USENIX Security Symposium*.

[24] O Hammami. 1995. Taking into account access patterns irregularity when compressing address traces. In *Southeastcon'95. Visualize the Future., Proceedings., IEEE*. IEEE, 74–77.

[25] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108.

[26] Mark D Hill. 1998. DINERO IV trace-driven uniprocessor cache simulator. *http://www. cs. wisc. edu/˜ markhill* (1998).

[27] Eric E Johnson and Jiheng Ha. 1994. Lossless address trace compression for reducing file size and access time. In *International Phoenix Conference on Computers and Communications, IEEE Press, Los Alamitos, CA, USA*. 213–219.

[28] Ajay Joshi, Lieven Eeckhout, and Lizy John. 2008. The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop*. 1–11.

[29] J Yi Joshua, Resit Sendag, Lieven Eeckhout, Ajay Joshi, David J Lilja, and Lizy K John. 2006. Evaluating benchmark subsetting approaches. In *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 93–104.

[30] James R Larus. 1999. Whole program paths. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 259–269.

[31] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2007. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 106–115.

[32] Yun Li, Jian Ren, and Jie Wu. 2012. Quantitative measurement and design of source-location privacy schemes for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems* 23, 7 (2012), 1302–1311.

[33] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. 2006. l-diversity: Privacy beyond k-anonymity. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 24–24.

[34] Stephen McCamant and Michael D Ernst. 2008. Quantitative information flow as network flow capacity. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 193–205.

[35] Pierre Michaud. 2009. Online compression of cache-filtered address traces. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 185–194.

[36] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust deanonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 111–125.

[37] Craig G Nevill-Manning and Ian H Witten. 1997. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference, 1997. DCC'97. Proceedings*. IEEE, 3–11.

[38] Catherine Mills Olschanowsky, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2009. PSnAP: Accurate Synthetic Address Streams through Memory Profiles.. In *LCPC*. Springer, 353–367.

[39] Mark Oskin, Frederic T Chong, and Matthew Farrens. 2000. *HLS: Combining statistical and symbolic simulation to guide microprocessor designs*. Vol. 28. ACM.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 1–20.

[41] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct

(2011), 2825–2830.

[42] Juan Rodriguez-Rosell. 1976. Empirical data reference behavior in data base systems. *Computer* 9, 11 (1976), 9–13.

[43] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. 2012. Phase guided profiling for fast cache modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 175–185.

[44] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. *ACM SIGPLAN Notices* 39, 11 (2004), 165–176.

[45] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News* 30, 5 (2002), 45–57.

[46] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. 2003. Discovering and exploiting program phases. *IEEE micro* 23, 6 (2003), 84–93.

[47] Latanya Sweeney. 2000. Simple demographics often identify people uniquely. *Health (San Francisco)* 671 (2000), 1–34.

[48] Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.

[49] Dominique Thiebaut, Joel L. Wolf, and Harold S. Stone. 1992. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on computers* 41, 4 (1992), 388–410.

[50] Mustafa M Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. 2006. PMaC Binary Instrumentation Library for PowerPC/AIX. In *Workshop on Binary Instrumentation and Applications*.

[51] New York Times. 2006. A Face Is Exposed for AOL Searcher No. 4417749. https://www.nytimes.com/2006/08/09/technology/09aol.html

[52] Luk Van Ertvelde and Lieven Eeckhout. 2008. Dispersing proprietary applications as benchmarks through code mutation. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 201–210.

[53] Jonathan Weinberg and Allan Snavely. 2008. Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications. In *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway*.

[54] Jonathan Weinberg and Allan Edward Snavely. 2008. Accurate memory signatures and synthetic address traces for HPC applications. In *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 36–45.