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.

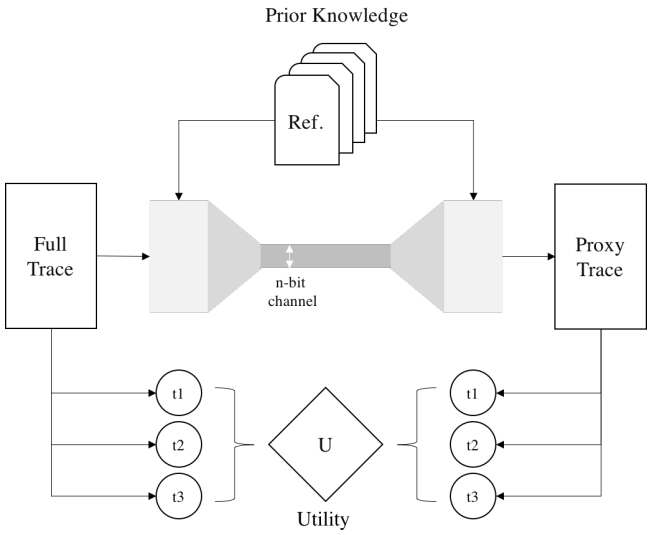


Figure 1. Forcing a trace through a channel with a capacity of only a few bits bounds the amount of sensitive data shared. While any public information such as prior non-private traces can be used in the creation of the code, the trace to be coded must not be known to the receiver. The objective then 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.

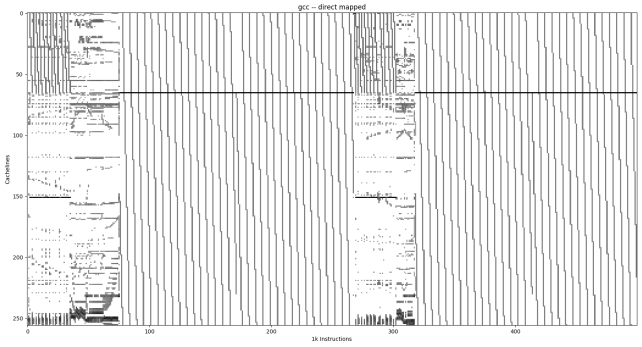


Figure 2. The modulo-memory access heatmap for gcc. The heatmap is an $N \times M$ sized graph, where N is some high power of 2 and M is the number of 10000 instruction windows in the trace. These *modulo-memory access heatmaps* illustrate patterns that exist within program executions, and give us a 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].

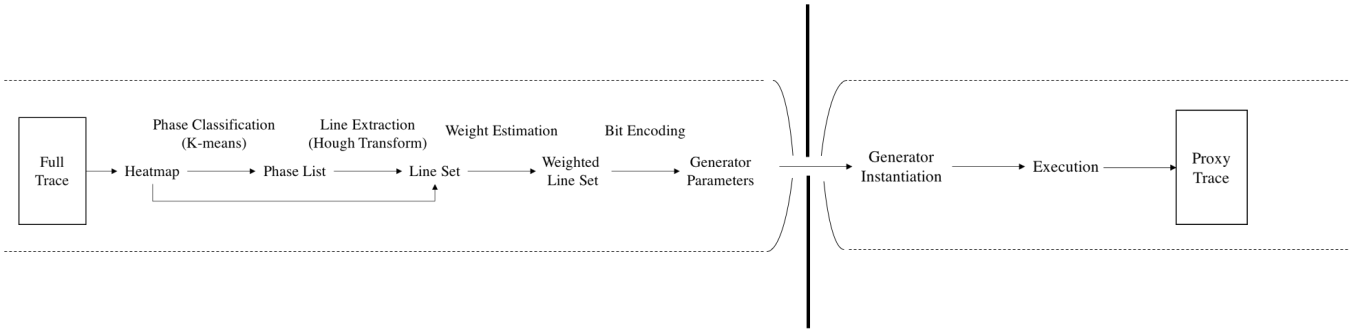


Figure 3. Pipeline for our signal processing approach to trace-wringing for proxy trace generation. The problem of sharing information can be described with two subsystems; at the trace-wringing end, we find parameters that will accurately generate the trace at the generator subsystem end. The goal is 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: $r = x \cos\theta + y \sin\theta$.

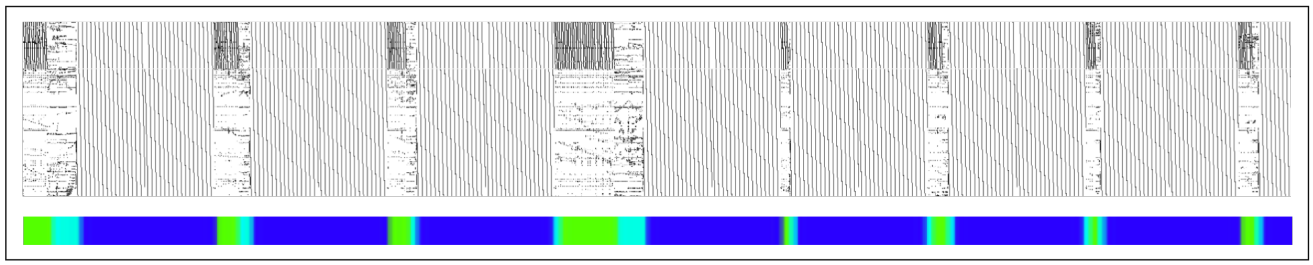


Figure 4. Phases visible in the trace generated by gcc after k-means clustering. Each of the 3 colors in the bottom marks a unique phase in the trace. Note, importantly, that phases reoccur over time.

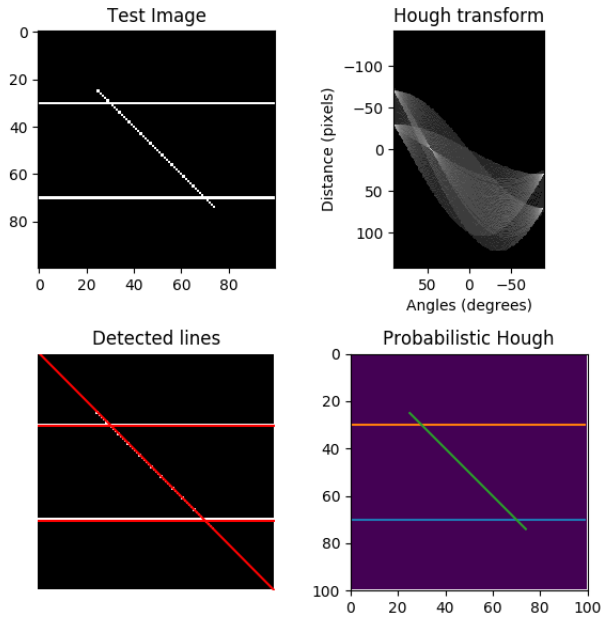


Figure 5. We capture information about lines we observe in trace heatmaps using the Hough Transform. Here, we demonstrate its working. The points on the test image are surveyed for parameters in the polar coordinate space described as the Hough Transform. The intersections describe the parameters of 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

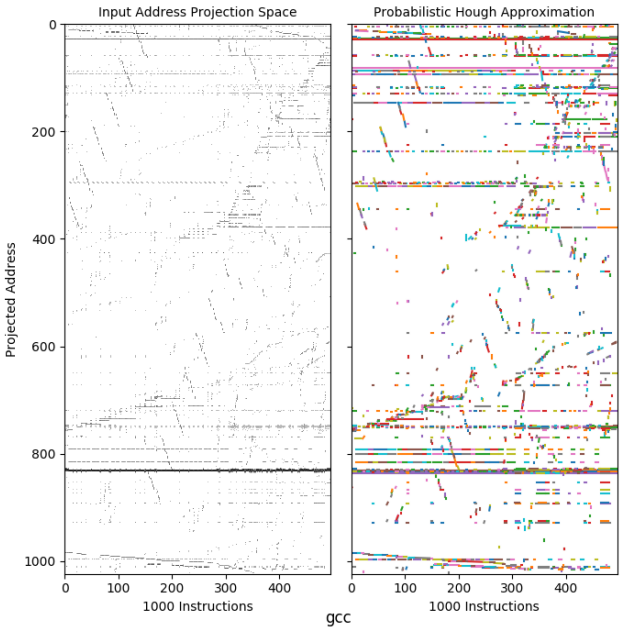


Figure 6. Producing probabilistic Hough lines on top of the heatmap of the SPEC2006 benchmark, gcc. The colors are used to indicate distinct lines produced by the decomposition.

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.

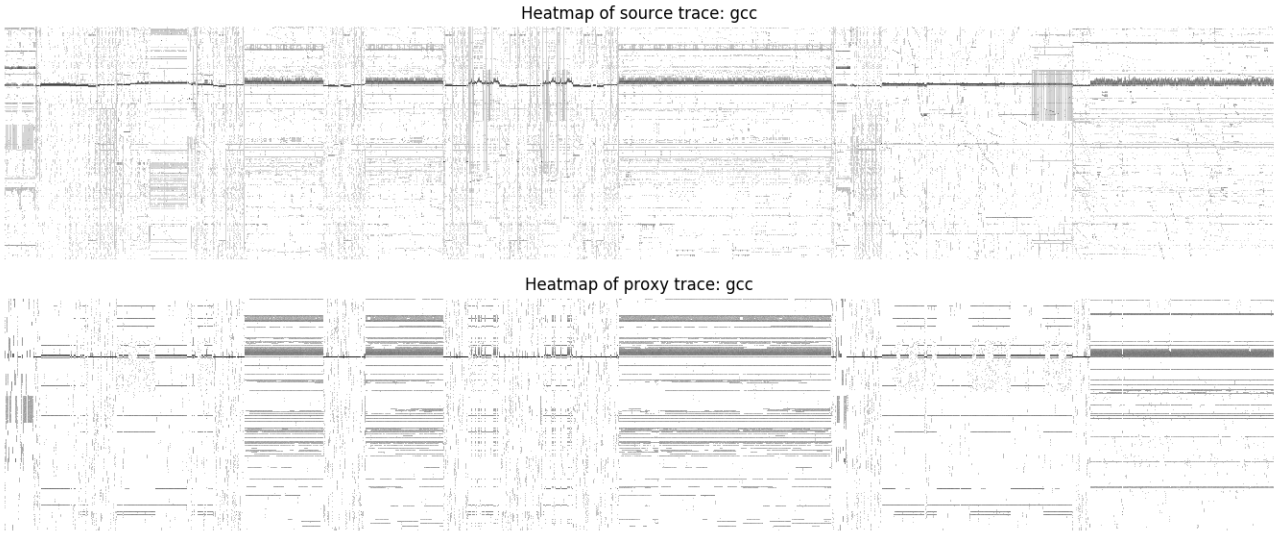


Figure 7. Heatmap for the original gcc trace and the trace-wrung proxy generated for gcc trace from the wrapped address space. Each pixel corresponds to one wrapped address at one time step. The darker the pixel, the more times that address is accessed during that time step.

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.

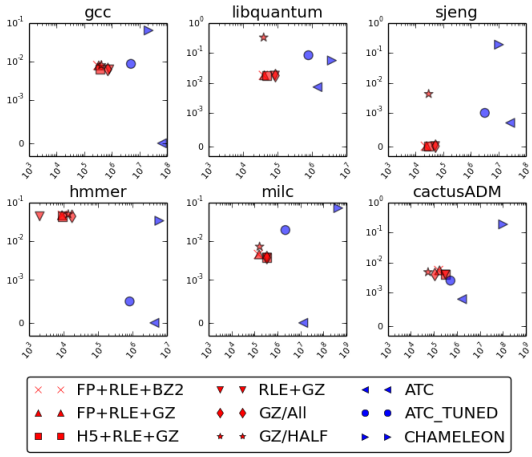


Figure 8. Breakdown of trace-wringing pipelines and comparison against state-of-the-art compression and synthetic trace generation techniques in the bit-error space. The x-axis represents *number of bits* transmitted, y-axis represents the *geometric-mean* of error in miss rate. Per workload, we mark 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.

Charm: A Language for Closed-form High-level Architecture Modeling

Weilong Cui*, Yongshan Ding†, Deeksha Dangwal*, Adam Holmes†, Joseph McMahan*,
Ali Javadi-Abhari‡, Georgios Tzimpragos*, Frederic T. Chong† and Timothy Sherwood*
*University of California, Santa Barbara
{cuiwl, deeksha, jmcmahan, gtzimpragos, sherwood}@cs.ucsb.edu
†University of Chicago
{yongshan, adholmes}@uchicago.edu, chong@cs.uchicago.edu
‡IBM Research
ali.javadi@ibm.com

***Abstract*—As computer architecture continues to expand beyond software-agnostic microarchitecture to data center organization, reconfigurable logic, heterogeneous systems, application-specific logic, and even radically different technologies such as quantum computing, detailed cycle-level simulation is no longer presupposed. Exploring designs under such complex interacting relationships (e.g., performance, energy, thermal, cost, voltage, frequency, cooling energy, leakage, etc.) calls for a more integrative but higher-level approach. We propose Charm, a domain specific language supporting Closed-form High-level ARchitecture Modeling. Charm enables mathematical representations of mutually dependent architectural relationships to be specified, composed, checked, evaluated and reused. The language is interpreted through a combination of symbolic evaluation (e.g., restructuring) and compiler techniques (e.g., memoization and invariant hoisting), generating executable evaluation functions and optimized analysis procedures. Further supporting reuse, a type system constrains architectural quantities and ensures models operate only in a validated domain. Through two case studies, we demonstrate that Charm allows one to define high-level architecture models concisely, maximize reusability, capture unreasonable assumptions and inputs, and significantly speedup design space exploration.**

***Keywords*-abstraction; modeling; DSL;**

I. Introduction

Computer architecture is evolving into a field asked to cover a tremendous space of designs. From the smallest embedded system to the largest warehouse-scale computing infrastructure, from the most well-characterized CMOS technology node to novel devices at the edge of our understanding, computer architects are expected to be able to speak to the non-orthogonal concerns of energy, cost, leakage, cooling, complexity, area, power, yield, and of course performance of a set of designs. Even radical approaches such as DNA-based computing [1] and quantum architectures [2], [3] are to be considered. While there are a great deal of well considered infrastructures to build around when detailed cycle-level simulation is required, for engineering questions that span multiple interacting constraints or to extreme scales the best approaches are more ad-hoc.

Careful application of detailed simulation can accurately estimate the potential of a specific microarchitecture, but exploration across higher level questions always involves some analytic models. For example, "given some target cooling budget, how much more performance can I get out of an ASIC versus an FGPA for this application given my ASIC will be 2 tech nodes behind the FPGA?" The explosion of domain-targeted computing solutions means that more and more people are being asked to answer these questions accurately and with some understanding of the confidence in those answers. While any Ph.D. in Computer Architecture should be able to answer this question, when you break it down, it requires a combination of a surprisingly complex set of assumptions. How do tech node and performance relate? What is the relationship between energy use and performance? ASIC and FPGA performance? Dynamic and leakage power? Temperature and leakage? Any result computed from these relationships will rely on the specific relationships chosen, on those relationships being accurate in the range of evaluation, on a sufficient number of assumptions being made to produce an answer (either implicitly or explicitly), and finally on that the end result be executable to the degree necessary to explore a set of options (such as for a varying parameter e.g., total cooling budget).

Such analysis today is not supported in any structured form. Typically it exists as a set of equations in an Excel spreadsheet or perhaps as a set of handwritten functions in a scripting language. Unfortunately, this comes with some issues. As simple as sets of mathematical relationships between quantities, the lack of a common engineering basis for these models have kept them from being swiftly and correctly constructed, understood and applied in guiding new system designs. Some models share a set of common relationships but they redefine those symbols and equations often with subtle differences that can be misleading if one is not careful. Some have implicit constraints on one or more architectural quantities which may lead to pitfalls if overlooked. Finally, one has to manually convert these mathematical equations to executable functions in order to evaluate the model and perform the design space exploration, which can be error-prone and inefficient.

To address these issues we design and explore a declarative domain specific language, Charm, to serve as a unified basis for the representation, execution, and optimization of closed-

form high-level architecture models. Charm provides a concise and natural abstraction to express architectural relationships and declare analysis goals. By combining symbolic manipulation, constraint solving, and compiler techniques, Charm bridges the gap between mathematical equations and executable, optimized evaluation functions and analysis procedures. The benefit of building and evaluating closed-form high-level architecture models using Charm is threefold:

Abstraction – Charm encapsulates a set of mutually dependent relationships and supports flexible function generation. It enables representation of architecture models in a mathematically consistent way. Depending on which metric the model is trying to evaluate, Charm can generate corresponding functions without requiring the user to re-write the equations. It also modulates high-level architecture models by packing commonly used equations, constraints and assumptions in modules. These architectural "rules of thumb" can then be easily composed, reused and extended in a variety of modelling scenarios.

Type Checking – Charm enables new static and run-time checking capabilities on high-level architecture models by enforcing a type system in such models. One example is that many architecturally meaningful variables have inherent physical bounds that they must satisfy; otherwise, although mathematically viable, the solution is not reasonable from an architectural point of view. With the type system built-in, Charm can dynamically check if all variables are within defined bounds to ensure a meaningful modelling result. The type system also helps prune the design space based on constraints, without which a declarative analysis might end up wasting a huge amount of computing effort in less meaningful sub-spaces.

Optimization – Charm opens up new opportunities for compiler-level optimization when evaluating architecture models. Although high-level architecture models are usually several orders of magnitude faster than detailed simulations, as the model gets complicated or is applied many times to estimate a distribution, it can still take a non-trivial amount of time to naively evaluate the set of equations every iteration. By expressing these complicated models in Charm, we are able to identify common intermediate results to hoist outside of the main design option iteration and/or apply memoization on functions.

Finally, and perhaps most importantly to the community, it promotes collaboration between application designers, computer architects, and hardware engineers because they can share and refine models using the same formal specification and a common set of abstractions.

We release Charm as an open-source tool on github[1] and we provide a wide collection of established architecture models for quick use/reference, including: the dark silicon model [4], a resource overhead model for implementing magic state distillation on surface code [5]–[7], mechanistic cpu models [8], [9], a TCAM power model [10], the LogCA model for accelerators [11], the adder/multiplier models from PyRTL [12], a widely-used CNN model [13], dynamic power and area models for NoC [14], specifications of Xilinx 7-series FPGA [15] and the extended Hill-Marty model [16].

To describe Charm we begin in Section II with a motivating example high-level model to show the problems with ad-hoc modelling in practice. Then we introduce the design of Charm in Section III followed by two case studies demonstrating the application and benefits of building closed-form high-level architectural models with Charm in Section IV. Finally we discuss the related works in Section V and conclude in Section VI.

II. Charm by Example

To understand Charm it is useful to have a running example. In this section, we present an implementation of the model and analysis from a well-cited study of dark silicon scaling [4]. After a brief review of the models, we show the complete code in Charm performing the same analysis of symmetric topology with ITRS technology scaling predictions. As we extend this model to cover more analysis provided in [4], it leads to a discussion of the potential issues with less structured approaches and highlights some of the features of the language that help architects avoid these pitfalls.

A. A Brief Review of the Dark Silicon Model

To forecast the degree to which dark silicon will become prevelent on CMPs under process scaling, Esmaeilzadeh et al. first construct three models: a device model (*DevM*), a core model (*CorM*) and a CMP model (*CmpM*). *DevM* is the technology scaling model relating *tech node* to *frequency scaling factor* and *power scaling factor*. It is a composite model combining a scaling prediction with a simple dynamic power model ($P = \alpha C V_{dd}^2 f$). *CorM* is the model relating *core performance*, *core power*, and *core area*. It is empirically deduced by fitting real processor data points. *CmpM* has two flavors which are essentially very different models: $CmpM_U$ and $CmpM_R$. $CmpM_U$ is an extension of the Hill-Marty CMP model [17] and $CmpM_R$ is a mechanistic model [18].

A composition of the three models is then used to drive the design space exploration. The authors combine *DevM* and *CorM* to look at *CorM* for different *tech node* and combine *DevM*, *CorM*, and *CmpM* to iterate over a collections of topologies, scaling predictions and core configurations. They then plot the scaling curves for the dynamic topology/$CmpM_R$ with both ITRS and a conservative scaling [19].

B. A Complete Charm Code Example

Listing 1 gives the complete code in Charm DSL to run the design space exploration with ITRS predictions on the symmetric topology (we later extend the analysis to other topologies and predictions in Section IV-A). At a high level, we can see that the code is split into three major components:

[1] https://github.com/UCSBarchlab/Charm.git

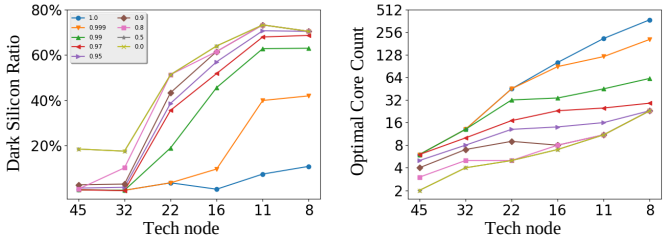


Fig. 1. Upper-bound ITRS scaling with symmetric topology.

type definition (Line 3-8[2]), model specification (Line 11-56) and analysis declaration (Line 59-66).

Specifically, we first define commonly used domains as Charm types on the architectural quantities we care about (Line 3-8). For example, the parallelism parameter in the model has a physical meaning of the proportion of the algorithm that can be parallelized and it naturally falls between $[0, 1]$. We thus define a type *Fraction* to encapsulate this domain constraint. While this is a simple example, more complex constraints are possible.

We then formally specify the three models (*DevM*, *CorM*, *CmpM*) to evaluate (Line 11-56). Taking the *ExtendedPollacksRule* model (Line 34-41) as an example, it declares upfront all the architectural quantities that are involved in the model (e.g., *ref_core_area* which is the core size at the reference technology node), their types (e.g., *ref_core_area* is a real number on the positive domain) and the relationships between the architectural quantities, e.g., $area = 0.0152perf^2 + 0.0265perf + 7.4393$ (the constants come directly from the original dark silicon paper [4]).

Once the models are defined, it is straightforward to declare the analysis in Charm (Line 59-66). One simply selects the given models in the study, supplies the inputs and specifies the target metrics to explore. For example, in this case, we select *ITRS*, *ExtendedPollacksRule* and *SymmetricAmdahl* models (Line 59), provide values such as the area (Line 60) and power (Line 61) constraints, and finally tell Charm what quantities we care to explore, in this case *speedup*, *dark_silicon_ratio* and *core_num* (Line 66).

C. Unstructured High-level Architecture Modeling Pitfalls

Building and executing an architectural model with an unstructured approach (e.g., in a spreadsheet or some general purpose scripting language) is clearly possible[3], but the lack of a common abstraction introduces some issues as one tries to scale the analysis. Each additional interacting component is a set of new opportunities to make an uncaught mistake.

The degree to which these mistakes end up in the final model (and the amount of effort required to make sure it is mistake-free) is a function of the degree of composability, reusability, consistency and completeness checking supported

[2]Line numbers in Section II all refer to Listing 1 unless otherwise specified.

[3]With all the potential issues, unstructured methods in architectural modeling may not be as correct as one tends to believe [20], [21].

```
1  # Type definitions.
2  # A real number greater than 0.
3  typedef R+ : Real r
4      r > 0
5
6  # A real number between [0, 1].
7  typedef Fraction : Real f
8      0 < = f, f < = 1
9
10 # Simple Fit of the ITRS Scaling (DevM).
11 define ITRS:
12     ref_tech_node : R+ as ref_t
13     ref_core_performance : R+ as ref_perf
14     ref_core_power : R+ as ref_power
15     ref_core_area : R+ as ref_area
16     tech_node : R+ as t
17     core_performance : R+ as perf
18     core_power : R+ as power
19     core_area : R+ as area
20     perf_scaling_factor : R+ as a
21     power_scaling_factor : R+ as b
22     ref_t = 45
23     perf = a * ref_perf
24     power = b * ref_power
25     area / t**2 = ref_area / ref_t**2
26     a = piecewise((1., t=45),   (1.09, t=32),
27                   (2.38, t=22), (3.21, t=16),
28                   (4.17, t=11), (3.85, t=8))
29     b = piecewise((1., t=45),   (0.66, t=32),
30                   (0.54, t=22), (0.38, t=16),
31                   (0.25, t=11), (0.12, t=8))
32
33 # Pollock's Rule Extended with Power (CorM).
34 define ExtendedPollacksRule:
35     ref_core_performance : R+ as perf
36     ref_core_area : R+ as area
37     ref_core_power : R+ as power
38     area = 0.0152*perf**2 + 0.0265*perf + 7.4393
39     power = 0.0002*perf**3 + 0.0009*perf**2
40             + 0.3859*perf - 0.0301
41     perf < 50
42
43 # Amdahl's Law under Symmetric Multicore (CmpM_U).
44 define SymmetricAmdahl:
45     speedup : R+ as sp
46     core_performance : R+ as perf
47     core_area : R+ as a
48     core_power : R+ as power
49     core_num : R+ as N
50     chip_area : R+ as A
51     thermal_design_power : R+ as TDP
52     fraction_parallelism : Fraction as F
53     dark_silicon_ratio : Fraction as R
54     sp = 1 / ((1 - F) / perf + F / (perf * N))
55     N = min(floor(A / a), floor(TDP / power))
56     R * A = A - N * a
57
58 # Assumptions are now explicit and composable.
59 given ITRS, ExtendedPollacksRule, SymmetricAmdahl
60 assume chip_area = 111.0
61 assume thermal_design_power = 125.0
62 assume fraction_parallelism = [0.999, 0.99, 0.97,
63                                0.95, 0.9, 0.8, 0.5]
64 assume tech_node = [45, 32, 22, 16, 11, 8]
65 assume ref_core_performance = linspace(0, 50, 0.05)
66 explore speedup, dark_silicon_ratio, core_num
```

Listing 1. Dark silicon analysis on symmetric topology with ITRS scaling.

by the tool. It is easiest to see this if we talk specifically again about the code of our example dark silicon analysis.

We first note that, although clearly defined conceptually, the three models needed are each of a different *form*: *DevM* is essentially a table of different scaling factors, *CorM* is an empirical set of points and a regression curve and *CmpM* is in the form of mathematical equations relating a set of high-level architectural quantities. Furthermore, even if they were of the same form, they are not "functions" but rather a set of mathematical *relationships*. The distinction is quite important. With traditional lvalue / rvalue style assignments (common to both functions and spreadsheets) you end up with four issues:

Composition: It is hard to link the models' I/O together or even check if the models can be connected properly at all. Architectural models usually are connected to each other through some common system parameters or physical quantities. In this example, to do the dark silicon analysis, one needs to take scaling factors from tables in *DevM*, pass them as inputs to *CorM*, apply the values and re-fits the curve for different *tech node*, after which one then has to sample from the two Pareto curves in *CorM* and supply the samples to $CmpM_U$ for final evaluation. This chain of data movement and dependency is not explicitly exposed by the models, and it takes some effort to understand how these models link together. This issue of mismatched form is even more acute when one wishes to switch out the *CmpM* core model with the $CmpM_R$ core model because $CmpM_R$ takes a completely different set of inputs. With unstructured methods, one has to explicitly program these connections typically by function call chains. With Charm, one simply specifies all variables upfront within each model, and as long as the full variable names are consistent, Charm "wires up" the models through these channelling I/O variables. Perhaps most importantly, Charm throws an error when the models cannot be properly linked. For example, if one forgets to provide values for technology node (Line 64), Charm will complain that too many variables are free, or if the scaling model is about transistor rather than processor core, as long as the variables are properly named (e.g., one does not name transistor performance as *core* performance), Charm will capture this mismatch and warn that the models cannot be connected.

Restructuring and Reorientation: The models cannot be evaluated in a flexible way. Even though the model is a relationship between quantities, in spreadsheets or scripting languages one has to implement the evaluation as functions with fixed arguments. In this example, one typically codes up to evaluate the *speedup* from given value of *core performance*. If the control quantity is changed to another, say *core area*, one has to fix the code. An even worse, and probably more interesting, case is when the control becomes the one under investigation, i.e., the input/output of the functions are reversed. In our example here, it happens when one wishes to explore the core count constraint given a target dark silicon ratio. There is no easy way for ad hoc methods to deal with this kind of flexibility but to completely reprogram. While in Charm, models *are* interpreted as a set of *mutually* dependent relationships without a fixed direction, and Charm runtime will generate the needed functions based on the provided controls and the quantities to explore.

Reasoning under Uncertainty: Architectural models usually involve some uncertainties [16], such as how technology may scale over the next 10-15 years. It is natural for computer architects to first evaluate the model with some concrete values (e.g., the scaling factors in Line 26, 29) and then model the uncertain quantity as some distribution, e.g., Gaussian distribution, as in our case studies in Section IV. It requires non-trivial programming effort with spreadsheets and scripting languages to support uncertain random variables. Charm supports different forms of input values such as scalars, vectors as well as distributions to ease architectural exploration.

Exploration: The analysis procedure is often coupled with the model definition. A common practice for computer architects is to explore the design space by iterating over a set of design options or different values for some system configuration knobs. With the high-level models, architects usually write imperative instructions to iterate over specific variables, and when the iterative variable changes to another, it quickly becomes tedious and error-prone to break and reconstruct the many-fold nested *for* loops. Charm decouples the model specification (Line 11-56) from the analysis procedure declaration (Line 59-66). Such iterations over input values are declarative and transparent (as opposed to writing *for* loops imperatively) by simply providing a list of values as inputs (Line 62, 64 and 65) in Charm.

Secondly, computer architectural quantities often have certain physical meanings. For example, *core performance* typically cannot be negative. A potential issue with unstructured methods is that these boundaries are usually only programmed ad hoc in spreadsheets or scripting languages. A negative *core performance* may be totally *mathematically* valid and *will* lead to meaningless misleading result if not captured in the unstructured implementation. This issue is even more likely to occur in the following two cases.

Implicit Domain Constraints: Architectural models typically have their range of operation. Aside from the physical constraints, implicit domain constraints also come from how the model is built at first place. In the dark silicon example, the normalized performance of the real data points that the authors used to generate the *CorM* is in the range of $(0, 50)$. Even though one can argue that a core with *normalized performance of 100* generally follows that regression but the result derived from that is much less accurate and trusted. This type of constraints are at most times only implicitly conveyed through the model building process, where it leads to a potential pitfall when the model is reused, especially when one only tries to interpret and re-implement the model from natural language descriptions (like in a published paper). While Charm encourages model builders to put in these implicit constraints explicitly as constraints built in the model specifications, e.g., Line 41. Charm will automatically check to see if these

$$var, rn, tn \in Name \qquad rel \in Relation$$
$$val \in Value$$
$$p \in Program := \overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var}$$
$$td \in TypeDefinition := \textbf{typedef}\ tn\ \overrightarrow{rel}$$
$$rdef \in RuleDefinition := \textbf{define}\ rn\ \overrightarrow{rdecl}$$
$$rdecl \in RuleDeclaration := var\ tn \mid rel$$
$$a \in AnalyzeStatement := \textbf{given}\ \overrightarrow{rn} \mid \textbf{assume}\ \overrightarrow{asmt}$$
$$asmt \in Assignment := var = val$$

Fig. 2. Abstract syntax of charm. A program is a sequence of type definitions, rule definitions, analysis statements, and a list of variables to explore. Relations are atomic with respect to the semantics; they use the syntax and semantics of the backend solver. They use the standard arithmetic and comparison operators, and allow lists, tuples, and real numbers as possible values.

constraints are violated during evaluation.

Unbounded Distributions: Many architectural quantities follow normal distribution such as *core frequency* due to process variability [22]–[24]. When using these types of unbounded distributions, it sometimes violates the physical constraints of the quantity (*frequency* must be positive). In unstructured modeling, this check is completely ad hoc and, if overlooked, will lead to meaningless results. With Charm, this issue is automatically handled by the type checker, as long as one specifies a correct type for the quantity, e.g., *frequency : R+*.

Last but not least, the design space to cover is typically huge with high-level models. In the dark silicon model, the authors explore a hundred core configurations for each combination of a scaling trend in *DevM* and a CMP model from $CmpM_U$ or a workload with $CmpM_R$. The models are often to be evaluated hundreds of thousands, if not millions, of times which will take a non-trivial amount of time. It only becomes worse when one tries to evaluate models with uncertainties [16]. Without a structured system, a quick spreadsheet or naive prototyping will end up with unacceptable performance when the problem is scaled up and the burden of optimization falls upon the model builders and others who wish to use existing models through re-implementation. As we show in Section IV, with the invariant hoisting and memoization techniques, Charm greatly speeds up the exploration without additional effort from the model builders.

III. Charm Design

Charm provides a simple domain specific modeling language to express both closed-form models and the design space exploration dimensions. The DSL has an easy-to-use Python-like syntax. In terms of mathematical expressiveness, Charm supports all common closed-form algebra that computer architects often resort to, including linear algebra like polynomials and simple non-linear algebra like exponentiation. Basic non-closed-form functions like summation and product are also supported. To enhance the design space exploration to uncertain domains, Charm also supports distributional values to be set and propagated through the models transparently. Once written in Charm DSL, the interpreter is able to transform the mathematical relationships and constraints into a series of data-flow graphs for fast evaluation. A type system is applied to make sure all architecturally meaningful quantities operate in the correct domain. Charm also optimizes the design space exploration procedure using compiler techniques to eliminate redundant computation. Figure 4 graphically shows the interpretation process.

In this section, we first describe the abstractions Charm provides and formalize the syntax and semantics of Charm DSL. We then articulate the internal design of the interpreter and how type checking, definability checking, evaluation and optimization are done in Charm.

A. Language Abstractions

Charm provides a common layer with three key abstractions to address all the potential issues in Section II-C. In Charm DSL, five keywords are reserved to express three abstractions: **types**, **models** and **analysis**.

Keyword *typedef* translates into the first abstraction: **type**. The type system is designed to be simple but useful: each type is essentially a base type with constraints, e.g., *R+* is defined as a positive number of base type *real* in Listing 1 Line 3-4. There are only two base types, *Real* and *Integer* standing for real numbers and integer numbers respectively. Internally, real numbers are represented by *float* and integers by *int*.

The second key abstraction is **model**. Keyword *define* constructs a model. A model specification in Charm encapsulates the following three pieces in a high-level architecture model.

A set of variables. Each variable has a universally consistent full name. Each variable also has a local short name (optional), as well as explicitly declared types. The short names only live within the definition and the full names are exported to other models and the analysis.

A set of equations. Equations define mathematical relationships between variables using either their full or short names (e.g., Listing 1 Line 54-56). Both linear and nonlinear systems are present in the common architectural models we care about. The general problem of trying to determine the definability of and solving such systems is theoretically hard and beyond the scope of this work. Given the limitations of the solving capabilities of the back-end solvers, some very complicated non-linear equations cannot be symbolically solved (e.g., solve for x in $y = (a^{1/x})^{2^x}$). Fortunately, we find that most models computer architects care about (even complicated as quantum computing in Section IV-B) are well within the limit. Equations can also bind variables to constant quantities as assumptions defined within the model specification (e.g., $kBoltzmann = 8.6173303 \times 10^5$).

A set of constraints. Inequalities define additional constraints on variables or expressions (e.g., Listing 1 Line 41). The difference between equations and constraints in Charm is that equations can be value generative if all but one variable are

$$C, D, E, \Omega \in RelationSet \quad \Gamma \in TypeEnvironment = Name \to RelationSet$$
$$\Theta \in RuleEnvironment = Name \to RelationSet \quad \mu \in VariableMap = Name \to Value$$

$$\frac{C = \left\{c \mid c \in \overrightarrow{rel}\right\}}{\textbf{typedef}\ tn\ \overrightarrow{rel} \Downarrow_T (tn, C)}\ \text{TYPEDEF} \qquad \frac{(\Gamma, rdecl_i) \Downarrow C_i \quad C = \bigcup C_i \quad i \in 1..|\overrightarrow{rdecl}|}{\left(\Gamma, \textbf{define}\ rn\ \overrightarrow{rdecl}\right) \Downarrow_R (rn, C)}\ \text{RULEDEF} \qquad \frac{\Gamma(tn) = C}{(\Gamma, var\ tn) \Downarrow C\,[var/tn]}\ \text{RD-VAR}$$

$$\frac{}{(\_, rel) \Downarrow \{rel\}}\ \text{RD-REL} \qquad \frac{C_i = \Theta(rn_i) \quad C = \bigcup C_i \quad i \in 0..|\overrightarrow{rn}|}{\left(\Theta, \textbf{given}\ \overrightarrow{rn}\right) \Downarrow_A C}\ \text{GIVEN} \qquad \frac{}{\left(\_, \textbf{assume}\ \overrightarrow{asmt}\right) \Downarrow_A \left\{e \mid e \in \overrightarrow{asmt}\right\}}\ \text{ASSUME}$$

$$\frac{\begin{array}{c}\texttt{Ext}(x) = \emptyset \bigvee \texttt{Ext}(y) = \emptyset \bigvee \texttt{Ext}(x) = \texttt{Ext}(y), \forall x, y \in \texttt{vars}(rel) \\ \omega = \left\{\alpha(a) \mid a \in \bigcup \texttt{Ext}(b_i), \forall b_i \in \texttt{vars}(rel)\right\} \quad \alpha(a) = rel[x.a/x, \forall x \in \texttt{vars}(rel)]\end{array}}{rel \Downarrow_M \omega}\ \text{MULTI-INSTANCE}$$

$$\frac{\begin{array}{c}\Gamma(tn_i) = C_i \text{ where } td_i \Downarrow_T (tn_i, C_i) \qquad \Theta(rn_j) = D_j \text{ where } \left(\Gamma, rdef_j\right) \Downarrow_R (rn_j, D_j) \\ \Omega = \bigcup E_k \text{ where } (\Theta, a_k) \Downarrow_A E_k \qquad \Omega' = \bigcup\{\omega \mid rel \Downarrow_M \omega \bigwedge rel \in \Omega\} \qquad \texttt{isConsistent}(\Omega') \\ \texttt{isFullyDetermined}\left(\Omega', \overrightarrow{var}\right) \qquad \mu = \texttt{SOLVE}\left(\Omega', \overrightarrow{var}\right) \qquad i \in 1..|\overrightarrow{td}| \qquad j \in 1..|\overrightarrow{rdef}| \qquad k \in 1..|\overrightarrow{a}|\end{array}}{\overrightarrow{td}\ \overrightarrow{rdef}\ \overrightarrow{a}\ \textbf{explore}\ \overrightarrow{var} \Downarrow_P \mu}\ \text{PROGRAM}$$

Fig. 3. Operational semantics of Charm. Relations are here taken as atoms; they use the semantics of the backend solver engine. An overhead arrow indicates a sequence of one or more elements. $C[x/y]$ indicates to substitute all instances of y in C with x. `vars` returns the names of all variables used in the relation set, while `Ext` returns all extensions of a variable (portion of the name appearing after a dot when multi-instanced). `isConsistent` ensures the relation set is consistent. `isFullyDetermined` ensures the relation set is fully determined with respect to $\overrightarrow{var}$. `SOLVE` is an instance of the backend solver; it returns a mapping of all specified variables to values (real numbers, lists, and tuples). TYPEDEF takes a type definition and returns a tuple with type name and relation set. RULEDEF takes a rule definition and the type environment and returns a tuple with rule name and relation set. RD-VAR takes a type rule declaration and the type environment and returns a relation set, where relations on the indicated type now apply to the indicated variable. RD-REL takes a relation rule declaration and returns the same relation in a set. GIVEN takes a **given** analyze statement and the rule definitions and returns the relation set of the indicated rule. ASSUME takes an **assume** analyze statement and returns a relation set of all the declared equalities. MULTI-INSTANCE takes a relation and returns a set of relations, where the original relation is duplicated once for each extension possessed by its variables, with the names of the variables replaced by their extended version (as discussed in section III-B). PROGRAM takes a program and returns a map for the list of exploration variables, mapping each to real numbers, lists, and tuples determined by the backend solver.

given, while constraints require all variables given during evaluation. Inequalities are by definition constraints and, when all variables are given, an equation is over-determined and turns into a constraint. We refer to the set of both equations and constraints as *relations*.

Charm DSL accepts different mathematically equivalent forms of relations, so that different modelers with different background expertise can write the math in the conventional way of their own fields and use other models directly as they are without rewriting.

The Charm DSL is strongly typed. The model abstraction enforces explicit type declaration to make sure there are not implicit assumptions about data types and domains across models.

Charm abstracts the common structure of an **analysis** with three keywords: *given*, *assume* and *explore*.

Before computation, *given* statement selects the model in the analysis. If multiple models are selected, they are linked together automatically by the interpreter. Full names of variables are used to connect each other across models.

Although in general, many algebra systems can be solved without additional inputs, for computer architecture models, at most times, some control quantities need to be given (e.g., design options like *core size* and system configurations like *cache associativity*) in order to solve for the quantities under investigation (e.g., *speedup* of a CMP). Keyword *assume* serves such purpose by differentiating assignment equal signs from mathematical equal signs inside model specification, i.e., *assume* statements are assignments much like in other programming languages while equations in model specification are merely mathematical relationships which do not imply a direction of data movement. Charm also constrains *assume* statements to be assignment with constants, i.e., they can only be used to express external inputs to the model rather than defining additional relations outside of the model specification.

Charm supports both scalar and vector value assignments, as well as random variable of commonly used distributions, e.g., *Gaussian Distribution*.

Iteration is expressed in a Pythonic list-like syntax or functions that generates a list, e.g., *linspace*, and assigned

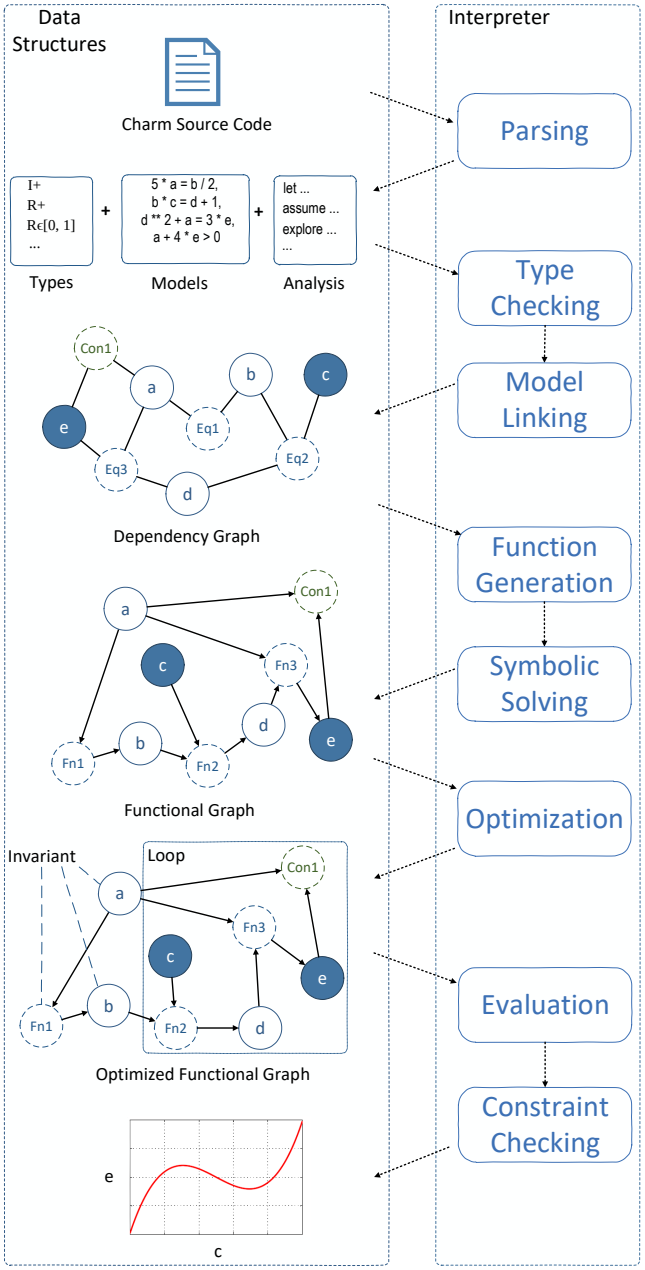


Fig. 4. Overview of Charm interpreter. The example system has 3 equations (Eq1, Eq2 and Eq3), 1 constraint (Con1) and 5 variables in which c is the iterative input. In this example, we are tying to explore the relationship between e and c given a. The parser takes Charm source code and breaks it into a set of types, a set of model definitions and a set of analysis statements. Charm then links types, models and assignments in a dependency graph after they go through the type checker. The graph then is fed to a function generator and a symbolic solver to convert it into a functional graph. The optimizer finally takes the functional graph and annotate it with hints for execution before it finally gets evaluated and checked against model constraints.

to some input variable just like a normal *assume* statement (e.g., Listing 1 Line 65). Charm handles iteration naturally by selecting combinations of all iterative input values non-repeatedly from their Cartesian space in a Gray code fashion. Two special cases are: a), if two or more input variables are dependent, they can be expressed like Python tuple assignment, e.g., $assume\ (tech\_node, freq\_scaling\_factor) = [(45, 1.), (32, 1.09)]$ and b), if a variable is indexed, it can be expressed using special "list" notation after its variable name, e.g., $assume\ L[] = [1, 2]$, which means $L[0] = 1$ and $L[1] = 2$. These notations become handy when we write the quantum models with Charm in Section IV-B.

Finally, an analysis is completed by specifying which quantities to solve for symbolically and evaluate using *explore*. Charm exploits a data-flow centric approach and builds a directed acyclic functional graph internally to propagate given values through linked models to the final responsive variables architects wish to explore.

Figure 2 gives the abstract syntax of Charm and Figure 3 formalizes the semantics.

B. Language Internals

In order to evaluate the models and optimize the evaluation logic, Charm uses two data-flow graph structures internally to represent and transform the computation. In this section, we first define the core graph data structures and then describe how we can perform type checking, function generation, evaluation and optimization with these graph structure.

Dependency Graph. A dependency graph is a bipartite graph $G =< V_{var}, V_{rel}, E >$, where:

- V_{var} is the variable node set in which every variable in the selected models is a vertex.
- V_{rel} is the relation node set and $V_{rel} = V_{eq} \cup V_{con}$, where V_{eq} is the set of vertices in which every equation in the selected models is a vertex; V_{con} is the set of vertices in which every constraint in the selected models is a vertex.
- E is the set of edges and there exists an edge between vertices in V_{var} and V_{rel} if and only if the variable name appears in the relation.

Functional Graph. A functional graph is a *directed acyclic* dependency graph D in which:

- Every node in V_{var} has at most 1 incoming edge, i.e., its in-degree being 0 or 1.
- Every node in V_{eq} has at most 1 outgoing edge, i.e., its out-degree being 0 or 1.
- Every node in V_{con} has no outgoing edge, i.e., its out-degree being 0.

Dependency graph building and static type checking. To build the dependency graph from the models, Charm performs a single scan over all relations in the models. It assigns a variable node to every variables with a unique full name (including variables automatically generated by multi-instancing) and an equation/constraint node to every equation/constraint. When creating relation node, Charm creates an edge between

the equation/constraint node to a variable node if the variable is used in the equation/constraint. Finally, Charm scans the analysis statements and marks variable nodes being assigned as input nodes.

Charm performs simple type checking both statically when building the dependency graph after parsing and dynamically when checking constraints at runtime. Static type checking is done by tracking the variable names and types when building the dependency graph. Each variable must be declared with an explicitly defined type. If a variable name is used by two or more relations, we check that their defined types are identical (both base type and constraints associated). Charm aborts execution and issues an error message for inconsistent types.

Relation Multi-instancing. When building a dependency graph, different variables sometimes follow the same mathematical relationships. An example is *core_performance.big* and *core_performance.small* defined in Listing 2 Line 5-6. Both of them follow equation in Listing 1 Line 23 when plugged in for evaluation. We discuss their physical meanings later in Section IV-A, but they are essentially two variables following the same mathematical relationship. We refer this behavior as "relation multi-instancing" and use the dot notation (a variable name and a name extension concatenated by dot, e.g., *core_area.big*) to invoke multi-instancing. Charm internally creates variable nodes and relation nodes for multiple instances with different name extensions. Figure 5 shows how these nodes in the dependency graph are created. The model is ill-defined if Charm fails to find extended input variables with consistent name extensions or discovers inconsistent name extension sets for different variables trying to invoke multi-instancing.

Functional graph building and function generation. After building the dependency graph G, the function converter tries to convert G into a functional graph F. If it can convert successfully, there is a viable solution when all equations or sets of equations can be solved and lambdified by the back-end symbolic solver, and therefore the models can be evaluated by Charm.

The function converter backtracks through G in a DFS manner and tries to label all the edges with a direction without introducing a conflict. A conflict occurs when an equation node has more than one outgoing edges or when an inequality node has any outgoing edge or when a variable node (excluding input nodes) does not have exact one incoming edge. If there is a successful labeling of all edges, Charm uses Sympy [25] as the back-end solver to convert all equations and constraints (all inequalities and equation nodes with an out-degree of 0 are considered as constraints at this point) into callable functions with inputs being the variables directly pointing to the equation/constraint and output being the variable pointed at by the equation node. As part of type checking, each variable node is also associated with the constraint from its type. These type constraints are also lambdified and evaluated during evaluation. The search space

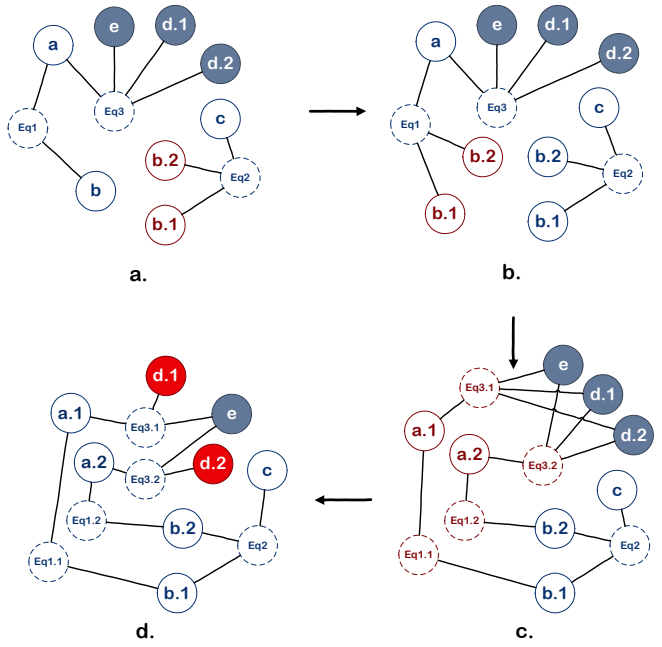


Fig. 5. Relation multi-instancing when generating dependency graph. a) The initial graph has extended names (b.1, b.2). b) Charm finds and splits the corresponding base name node. c) Charm propagates the multi-instancing, i.e., all nodes connected to the base name node (b) are also split. Then Charm merges names with same extension together. d) The multi-instancing ends with checking input nodes for identical name extensions and removing edges between non-consistent name extensions. In this case, it ends when the split process reaches d and e, successfully finds d.1 and d.2 which are extended names with consistent name extension set ({.1, .2} in this example) and removes the edges between (d.1, Eq3.2) and (d.2, Eq3.1).

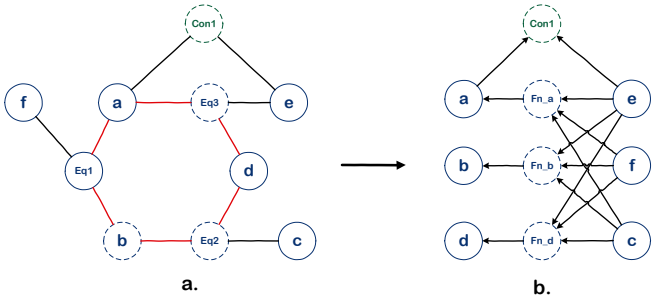


Fig. 6. Cycle elimination when generating functional graph. Equations in a cycle are solved at once and are replaced with three functions, each of which generates a different variable value.

for conversion is in practice greatly reduced by the following heuristics:

- All edges with one node being input node have fixed direction (from the input node).
- All edges with one node being a dangling variable node (variable node that has only one edge) have fixed direction (to the variable node).
- All edges with one node being a constraint have fixed direction (to the constraint node).

Cycle elimination. A functional graph F must be *acyclic*

in order to evaluate. However, when there are codependent equations, they form cycles. In case of a cycle, all equation nodes in the cycle must be solved altogether. We pass the equations in a cycle to the solver at once and then replace the cycle with pairs of function node and variable node, where each pair is a mapping between all inputs to the cycle (a dummy input node is created if there are no inputs from other parts of the graph to the cycle) and one variable node inside the cycle. Each function node generated by the cycle has one variable along the cycle as its output and all functions generated by the cycle are from the same set of equations, only with different variables as output. Figure 6 shows an example of cycle elimination in F.

Computational constraints. A special computational constraint is applied when building a functional graph: some mathematical operators are not reversible or have infinite solutions, such as $\sum$ and $\prod$, some are computationally hard for the solver, like solving x in $y = (a^{1/x})^{2^x}$. For the non-reversible equation, its direction is fixed, i.e. its edges have fixed direction not subject to the function converter.

Evaluation and constraint checking. Once we have a viable functional graph F, a feasible solution is to derive from all input nodes and propagate the given values by traversing F. Each following function/constraint node is transformed using higher-order functions to "remember" propagated partial values before all inputs are ready and it can be evaluated.

Optimization. Oftentimes architects explore the relationship between two variables by iterating over different input values. One simple yet effective optimization is invariant hoisting. With the functional graph structure, it is straightforward to optimize for invariant in Charm. From each iterative variable node, Charm simply traverse from that node, then all nodes that cannot be reached from the iterative input nodes are invariant to iteration over that input. In the simple illustrative example in Figure 4, c is iterative and a, b, *Fn1* are invariant because there are not paths from c to them.

Each function node also caches a mapping table between inputs and its output. Such memoization optimizes away unnecessary re-computation over same set of input values.

IV. Case Studies

In this section, we demonstrate the application of Charm using two case studies. In the first case study, we show the benefits of Charm by extending the dark silicon analysis with a different topology and a distribution of technology scaling. We also compare the execution times with and without optimization.

The second case study focuses more on the problem of modeling a critical resource in fault-tolerant quantum computing and performs exploration with varying physical error rate. Interestingly, when validating Charm results in the second case study, Charm helps find inconsistent model definition errors, which are silently propagated through by Mathematica [26] and would have led to incorrect results.

```
1  # Amdahl's Law under Asymmetric Multicore (CmpM_U).
2  define AsymmetricAmdahl:
3      speedup : R+ as sp
4      # here we need two types of perf, area, power
5      core_performance.big : R+ as big_perf
6      core_performance.small : R+ as small_perf
7      core_area.big : R+ as big_a
8      core_area.small : R+ as small_a
9      core_power.big : R+ as big_power
10     core_power.small : R+ as small_power
11     core_num : R+ as N
12     chip_area : R+ as A
13     thermal_design_power : R+ as TDP
14     fraction_parallelism : Fraction as F
15     dark_silicon_ratio : Fraction as R
16     sp = 1 / ((1-F)/big_perf + F/(N*small_perf+
       big_perf))
17     N = min(floor((A - big_a)/small_a),
18             floor((TDP - big_power)/small_power))
19     R * A = A - (N * small_a + big_a)
20     big_perf >= small_perf
21
22 given ITRS, ExtendedPollacksRule, AsymmetricAmdahl
23 assume ref_core_performance.big=linspace(0,50,0.05)
24 assume ref_core_performance.small=linspace
       (0,50,0.05)
```

Listing 2. Asymmetric model and the changes in code.

```
1  # Conservative scaling model (DevM).
2  define ConservativeScaling:
3      ...
4      a = piecewise((1., t=45),   (1.10, t=32),
5                    (1.19, t=22), (1.25, t=16),
6                    (1.30, t=11), (1.34, t=8))
7      b = piecewise((1., t=45),   (0.71, t=32),
8                    (0.52, t=22), (0.39, t=16),
9                    (0.29, t=11), (0.22, t=8))
10
11 given ConservativeScaling, ExtendedPollacksRule,
       AsymmetricAmdahl
```

Listing 3. Conservative scaling and the changes in code.

A. Dark Silicon and Beyond

Listing 2 highlights all the changes that we need to implement in Charm to model and switch the DSE from symmetric topology to asymmetric. Note that in the asymmetric model, "relation multi-instancing" comes in handy when expressing two co-existing types of core. To switch the analysis, all we need to do is to change the models that are *given* (Listing 2 Line 22) and provide values for two types of core instead of one (Listing 2 Line 23-24). We also write a new constraint (Listing 2 Line 20) to specify the fact that the big core should have better performance than the small core.

It's even simpler to switch from ITRS scaling predictions to the conservative predictions [19]. Listing 3 shows all the changes needed. Figure 7 plots the resulting scaling trends for the asymmetric topology.

One interesting question one may ask is *"what if the actual technology scaling is somewhere in between the two predictions?"* We explore the design space with a distribution of scaling factors. We use a Gaussian distribution for the scaling factor, the mean of which being the average value

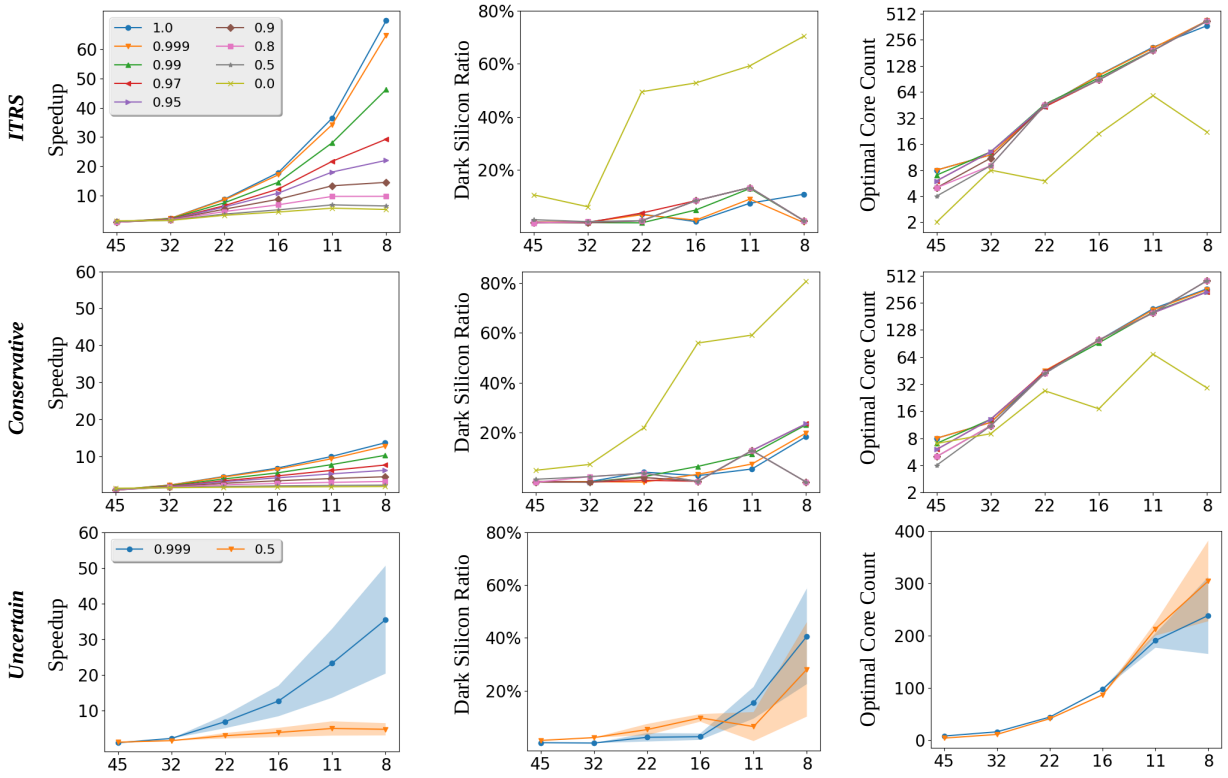


Fig. 7. Upper-bound scaling with asymmetric topology with tech node on x-axis. Note that the last figure of optimal core count has a linear-scale y-axis to better demonstrate the variance. For clarity we only plot two regions in the uncertain scaling results, but the trends for other f values are similar.

between the two extremities and the standard deviation being the difference between the mean and the extremities. Listing 4 shows the necessary changes in Charm code. It is important that although Gaussian distribution is not bounded, the scaling factors have a bounded domain. The type checking in Charm makes sure that the scaling factors a and b operate only in their defined domains (see Listing 1 Line 20-21), and the provided Gaussian distribution is converted to a truncated Gaussian distribution with the same mean and standard deviation within Charm. From Figure 7, we can see that with the technology scaling, the more parallel workload (with an f close to 1) shows more sensitivity towards technology uncertainties while the more serial workload is less sensitive to the changes in the core performance and power. Another probably even more interesting observation is that the optimal core count of the most performant configuration becomes very uncertain once we hit 11nm and beyond. The uncertainty grows sharply from 16nm to 11nm mainly because below 11nm, the CMP is mainly **area bounded**, and since the area scaling is certain (Listing 1 Line 25), it limits the amount of uncertainty that gets propagated to the optimal core count. Meanwhile, when the tech node scales to 11nm and beyond, the CMP becomes **power bounded** and is extremely sensitive to the power uncertainties propagated from the uncertainty of the power scaling factor.

Figure 8 shows the actual functional graph generated by Charm. In terms of execution performance, we compare

```
1  # Distributional scaling model (DevM).
2  define DistScaling:
3      ...
4    a = piecewise((1.,t=45),(Gauss(1.095,0.005),t=32),
5    (Gauss(1.785,0.595),t=22),(Gauss(2.23,0.98),t=16),
6    (Gauss(2.735,1.435),t=11),(Gauss(2.595,1.255),t=8)
       )
7    b = piecewise((1.,t=45),(Gauss(0.685,0.025),t=32),
8    (Gauss(0.53,0.01),t=22),(Gauss(0.385,0.005),t=16),
9    (Gauss(0.27,0.02),t=11),(Gauss(0.17,0.05),t=8))
10
11 given DistScaling, ExtendedPollacksRule,
       AsymmetricAmdahl
```

Listing 4. Uncertain scaling and the changes in code.

Charm execution to an unoptimized baseline in which all computation is re-done per iteration (no invariant hoisting nor memoization). For ITRS or conservative scaling with asymmetric topology (a design space of 150K design points), full-blown Charm finishes on average within 120.5s, while the unoptimized implementation uses 159.5s (1.3X speedup). For the uncertain scaling with a MC sample size of 200 (1̃ million design points), optimized Charm uses 1562.5s, and it takes 5703.1s for the baseline implementation (3.6X speedup) on a single Intel i7 core at 3.3GHz to finish.

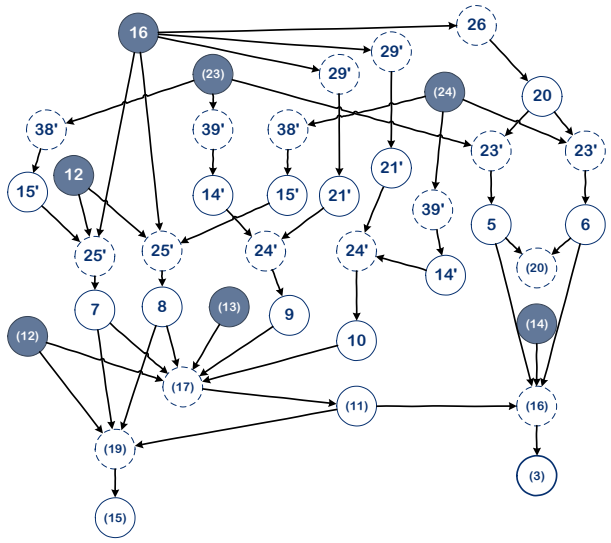


Fig. 8. Functional graph generated by Charm for asymmetric dark silicon model. Node labels correspond to line numbers in Charm source code. As we present the asymmetric model code separately from the rest, plain numbers correspond to lines in Listing 1, and numbers in parenthesis correspond to lines in Listing 2. Numbers with prime (e.g., 21') are cloned names/equations generated by Charm from the same line of code.

B. Surface Code Error Corrected Quantum Application and Architecture Co-optimization

In this section, a high level model for the resource overhead for implementing magic state distillation on surface code [5]–[7] is described and implemented within the Charm framework, which is used to pinpoint nontrivial interactions between fundamental system parameters.

For this study, we focus primarily on the Bravyi-Haah "$3k + 8 \rightarrow k$" procedure [5] augmented with the block-code protocol. By recursively stacking magic state distillation protocols in a tree-like fashion, one can generate arbitrarily high-fidelity magic states, which is required by a quantum program [7]. The space required by one round of Bravyi-Haah magic state distillation is given by the number of physical qubits required to run the circuit. Using block code, the procedure will consume $(3k+8)^{\ell-1}(6k+14)d^2$ physical qubits, where d is the surface code distance we are using.

Adding more factory capacity K results in more output magic state capacity (higher effective rate). However this also adds more components to the factory that may fail. In fact, a magic state factory has a **yield rate** proportional to the output capacity K that is caused by uncertainty in the success probability of the underlying Bravyi-Haah protocol. This yield rate scales as:

$$K_{\text{output}} = k^{\ell} \times \prod_{r=1}^{\ell} \left[1 - (3k+8)\epsilon_r\right] \tag{1}$$

where $\epsilon_r = (1+3k)^{2^r-1}\epsilon_{\text{in}}^{2^r}$, because each level of the process results in incrementally higher fidelity (i.e., lower error rate).

Given a T gate request distribution D representing a program, the number of iterations needed to distill is:

$$\sum_{t=0}^{T_{\text{peak}}} \left(s \cdot \sqrt{K'} + \sqrt{\frac{t - sK'}{2}} \cdot R\right) \cdot L_{\text{cp}} \cdot D[t], \tag{2}$$

where $s = \left\lfloor \frac{t}{K'} \right\rfloor$, and $R = \frac{7d+15}{24d\ell}$. All of these equations combine to form a high level space-time estimate of the resources required to execute a quantum application on a machine with a specified magic state distillation factory architecture.

Using Charm, we are able to analyze the underlying sensitivity of different magic state factory architectures to variations in the underlying error rate of the physical system. We examine two different design cases, one where the factory is designed assuming a 10^{-3} error rate, and one assuming a 10^{-5} error rate. Figure 9 illustrates that while the time-optimal factory does show a lower expected space-time volume, it also shows significantly higher uncertainty and spreads in performance values over the space-optimal factory. This design point clearly motivates that quantifying the uncertainty of a physical device is necessary to lead to risk-optimal system designs that perform well on a given system.

Charm is able to discover and quantify this trend with minimum effort, and allows for a quantitative analysis to be performed on these designs that will aid the construction of physical systems. Additionally, implementing this high level performance model in Charm allows for validation and more domain-specific error catching that previous implementations in Mathematica has been unable to catch. Specifically, a previous implementation has an incorrect parameter passed into a distance calculation function that Mathematica allowed to flow through. Charm is able to detect this error, warns that the models cannot be connected properly which helped correct the results of the model.

V. Related Work

A. Closed-form Architecture Models

Many of the recently developed high-level analytical models are conceptually inherent from the well known Amdahl's Law [27], which is often expressed as a closed-form performance model of parallel programs. The most well studied derivative is the multicore performance model by Hill and Marty [17]. A long line of research work using extensions of their closed-form model focus on different aspects of the system, including application [28], communication and synchronization [29], [30], energy and power consumption [4], [31], heterogeneity [11], [32], chip reliability [33], architectural risk [16] and so on. Our language consumes these models and provides a systematic way to establish new high-level models either by constructing new equations and constraints or reusing those from the above models.

Another set of analytical performance model is built directly from the mechanisms of the specific system [8], [9], [34]–[42]. These models usually rely on some simulations/hardware counter to collect the necessary inputs to their core closed-form equations. Our language can also express and manage these equations. Empirical modeling [43]–[51] is also used to

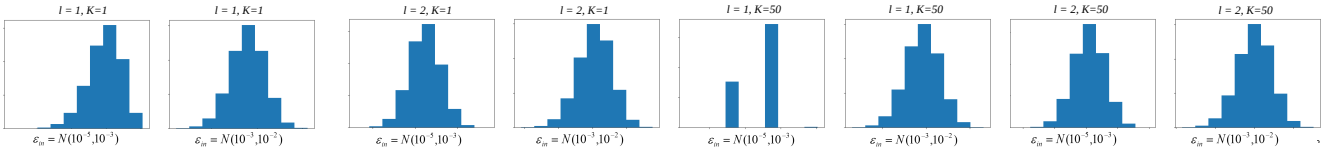


Fig. 9. Factories designed with an implied low physical error rate only require concatenation up to $\ell = 1$ level, while more pessimistic factories require $\ell = 2$. While larger factories with $K = 50$ consistently show lower mean space-time consumption, they also suffer from large performance uncertainty when the assumed design error rate varies.

discover correlation between two or more architectural quantities. They can usually be expressed as parametrized equations in closed-form, the resulting models of such empirical methods can also be managed by and benefit from Charm.

B. Systems and Languages Supporting Analytical Modeling

There exist systems and languages that support structured analytical modeling. Modelica [52] supports multi-domain analytical modeling with an emphasis on object-oriented model composition, but the connection of models need to be explicitly dictated and the design space exploration require user intervention, while Charm is more restricted and thus able to automatically link models and generate exploration loops. Aspen [53] provides a DSL to express application and an abstract machine organization in order to model performance. Palm [54] utilizes source code annotation to build analytical model for the application. LSE [55] is a fully concurrent-structural modeling framework designed to maximize reusability of components. There are also many other works in the field of HPC for automatic performance modelling extracting [56]–[58]. Most of these languages and systems serve a different purpose of expressing mapping between performance/power model and specific detailed application/architecture and are not well-suited for high-level analytical design space exploration. While Charm is tailored for structured yet flexible exploration of the interactions between architectural variables as well as their ramifications at a high level. There are also a few systems exploiting the power of symbolic execution for modeling [16], [20], but Charm provides more capabilities around formalizing, checking and evaluating the models. There also exits a tool [59] of the same name CHARM (Chip-architecture Planning Tool) which uses a knowledge-based scheme to ease high-level synthesis.

The internals of Charm resemble some of the data-flow centered programming languages in the field of incremental/reactive programming [60]–[64] but differ in that Charm is highly restrictive. The restrictiveness means that Charm is more of a modeling language rather than a programming language, i.e., Charm does not support general purpose structures like loops and function calls but supports a malleability useful for exploration (e.g., reversing input/output dependencies).

VI. Conclusion

Computer architecture is a rapidly evolving field. Complex and intricately interacting constraints around energy, temperature, performance, cost, and fabrication create a web of relationships. As we move toward more heterogeneous and accelerator-heavy techniques, our understanding of these relationships is more fundamental to the process of design and evaluation than ever before. Already today we are seeing machine learning [65], cryptography [66], and other fields attempting to pull architectural analysis into their own work – sometimes introducing serious bugs along the way. Architecture is now a field that is expected to make scientific statements connecting nanoscale device details to the largest warehouse scale computers and everything in between. Spanning these 11 orders of magnitude will require more complex analytic approaches to be used in tandem with the traditional simulation and prototyping tools that computer architects have long relied on.

Charm provides domain specific language support for architecture modeling in a way that leads to more flexible, scalable, shareable, and correct analytic models. While our language already supports symbolic restructuring, memoization, hoisting, and several optimization and consistency checks, Charm is merely the first step towards a more powerful and useful modeling language for computer architects. It is easy to imagine other useful additions in the future such as checks on the consistency of physical types (e.g., nJ versus pJ errors) or back-ends connecting models to non-linear optimizers. Most importantly though, by giving the sets of mutually dependent architectural relationships a common language, Charm along with the collection of established models have the potential to enable more complete and precise specification, easier composition, more through checking, and (most importantly) broader reuse and sharing of complex analytic models. Looking forward we see that tools such as this hold significant promise in enabling more collaborative and community driven efforts that make our best thinking on the future of architecture more readily and easily accessible to all that are interested.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grants No. 1740352, 1730309, 1717779, 1563935, 1444481, 1341058, 1730449, 1660686, Los Alamos National Laboratory and the U.S. Department of Defense under subcontract 431682 and gifts from Cisco Systems and Intel Corporation.

The authors would like to thank Michael Christensen for his help with formalizing the semantics and the anonymous reviewers for their invaluable feedback.

References

[1] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss, "A dna-based archival storage system," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 637–649. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872397

[2] A. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, "Optimized surface code communication in superconducting quantum computers," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 692–705. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123949

[3] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels, "An experimental microarchitecture for a superconducting quantum processor," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 813–825. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123952

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[5] S. Bravyi and J. Haah, "Magic-state distillation with low overhead," *Physical Review A*, vol. 86, no. 5, p. 052329, 2012.

[6] A. G. Fowler, S. J. Devitt, and C. Jones, "Surface code implementation of block code state distillation," *Scientific Reports*, vol. 3, p. 1939, jun 2013.

[7] J. O'Gorman and E. T. Campbell, "Quantum computation with realistic magic-state factories," *Physical Review A*, vol. 95, no. 3, p. 032338, 2017.

[8] M. Breughe, S. Eyerman, and L. Eeckhout, "A mechanistic performance model for superscalar in-order processors," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, April 2012, pp. 14–24.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 3:1–3:37, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1534909.1534910

[10] B. Agrawal and T. Sherwood, "Modeling tcam power for next generation network devices," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006, pp. 120–129.

[11] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 375–388. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080216

[12] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–7.

[13] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[14] A. B. Kahng, B. Li, L. S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 423–428.

[15] Xilinx, "7 series product tables and product selection guide," February 2018, online. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf

[16] W. Cui and T. Sherwood, "Estimating and understanding architectural risk," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017.

[17] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[18] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, Jan 2009.

[19] S. Borkar, "The exascale challenge," in *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, April 2010, pp. 2–3.

[20] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen, "Validating the unit correctness of spreadsheet programs," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 439–448. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999448

[21] S. G. Powell, K. R. Baker, and B. Lawson, "A critical review of the literature on spreadsheet errors," *Decis. Support Syst.*, vol. 46, no. 1, pp. 128–138, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.dss.2008.06.001

[22] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3–13, Feb 2008.

[23] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 504–514.

[24] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, July 2016.

[25] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016. [Online]. Available: http://www.sympy.org

[26] W. R. Inc., "Mathematica, Version 11.2," champaign, IL, 2017.

[27] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[28] X.-H. Sun and Y. Chen, "Reevaluating amdahl's law in the multicore era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, pp. 183–188, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.05.002

[29] L. Yavits, A. Morad, and R. Ginosar, "The effect of communication and synchronization on amdahl's law in multicore systems," *Parallel Computing*, vol. 40, no. 1, pp. 1 – 16, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113001324

[30] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 362–370. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816011

[31] D. H. Woo, D. H. Woo, D. H. Woo, D. H. Woo, H. H. S. Lee, H. H. S. Lee, H. H. S. Lee, and H. H. S. Lee, "Extending amdahl's law for energy-efficient computing in the many-core era," *Computer*, vol. 41, no. 12, pp. 24–31, Dec 2008.

[32] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 225–236. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.36

[33] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili, "Amdahls law for lifetime reliability scaling in heterogeneous multicore processors," in *The 2016 International Symposium on High-Performance Computer Architecture (HPCA-22)*, March 2016.

[34] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[35] ——, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer*

Architecture, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 280–289. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815998

[36] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 673–686.

[37] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 338–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006729

[38] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and mshrs," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, Nov 2008, pp. 59–70.

[39] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 216–226.

[40] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 83–94.

[41] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[42] A. A. Nair, S. Eyerman, J. Chen, L. K. John, and L. Eeckhout, "Mechanistic modeling of architectural vulnerability factor," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 11:1–11:32, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2669364

[43] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 7–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545217

[44] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881

[45] B. Lee and D. Brooks, "Statistically rigorous regression modeling for the microprocessor design space," in *ISCA-33: Workshop on Modeling, Benchmarking, and Simulation*, 2006.

[46] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168882

[47] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 262–271. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.26

[48] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of inference and learning for performance modeling of parallel applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 249–258. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229479

[49] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb 2007, pp. 340–351.

[50] B. C. Lee, J. Collins, H. Wang, and D. Brooks, "Cpr: Composable performance regression for scalable multiprocessor models," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 270–281. [Online]. Available: https://doi.org/10.1109/MICRO.2008.4771797

[51] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815967

[52] H. Elmqvist, S. Mattsson, H. Elmqvist, and D. Ab, "An introduction to the physical modeling language modelica," in *Proc. 9th European Simulation Sympossium ESS97, SCS Int*, 1997, pp. 110–114.

[53] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 84:1–84:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389110

[54] N. R. Tallent and A. Hoisie, "Palm: Easing the burden of analytical performance modeling," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597683

[55] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August, "The liberty simulation environment: A deliberate approach to high-level system modeling," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 211–249, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151690.1151691

[56] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "Exasat: An exascale co-design tool for performance modeling," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 2, pp. 209–232, May 2015. [Online]. Available: http://dx.doi.org/10.1177/1094342014568690

[57] S. R. Alam and J. S. Vetter, "A framework to develop symbolic performance models of parallel applications," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 320–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898699.1898852

[58] ——, "Hierarchical model validation of symbolic performance models of scientific kernels," in *European Conference on Parallel Processing*. Springer, 2006, pp. 65–77.

[59] K. H. Temme, "Charm: a synthesis tool for high-level chip-architecture planning," in *1989 Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1989, pp. 4.2/1–4.2/4.

[60] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991.

[61] L. Mandel and M. Pouzet, "Reactiveml: A reactive extension to ml," in *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '05. New York, NY, USA: ACM, 2005, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/1069774.1069782

[62] M. A. Hammer, U. A. Acar, and Y. Chen, "Ceal: A c-based language for self-adjusting computation," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 25–37. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542480

[63] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 320–331. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970298

[64] P. LeGuernic, T. Gautier, M. L. Borgne, and C. L. Maire, "Programming real-time applications with signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sep 1991.

[65] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[66] J. Alwen and J. Blocki, "Efficiently computing data-independent memory-hard functions," in *Annual Cryptology Conference*. Springer, 2016, pp. 241–271.

Analysis and Mitigations of Reverse Engineering Attacks on Local Feature Descriptors

Anonymous CVPR 2021 submission

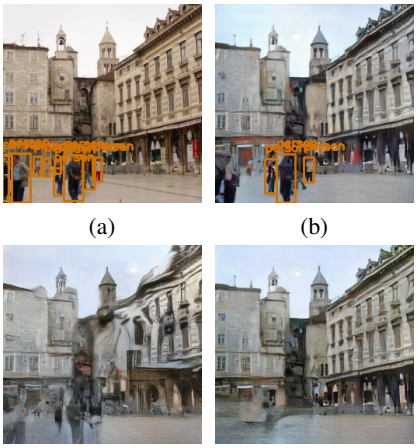
Paper ID

Abstract

As autonomous driving and augmented reality evolve, a practical concern is data privacy. In particular, these applications rely on localization based on user images. The widely adopted technology uses local feature descriptors, which are derived from the images and it was long thought that they could not be reverted back. However, recent work has demonstrated that under certain conditions reverse engineering attacks are possible and allow an adversary to reconstruct RGB images. This poses a potential risk to user privacy. We take this a step further and model potential adversaries using a privacy threat model. Subsequently, we show a reverse engineering attack on sparse feature maps and analyze the vulnerability of popular descriptors including FREAK, SIFT and SOSNet. Finally, we evaluate potential mitigation techniques that select a subset of descriptors to carefully balance privacy risk while preserving image matching accuracy; our results show that similar accuracy can be obtained when revealing less information.

1. Introduction

Privacy and security of user data has quickly become an important concern and a design consideration when engineering autonomous driving and augmented reality systems. In order to support machine perception stacks, these systems require always-on information capture. Most of these use-cases rely directly or indirectly on the data that originates from the user, i.e., RGB, inertial, depth, and other sensor values. Data assets are potentially rich in private information, but due to the compute power limitations on the device, they must be sent to a service provider to enable services such as localization, and virtual content. As a result, there is understandable concern that any data assets shared with a cloud service provider, no matter how well-trusted, can potentially be abused [5]. To enable augmented reality in practice, beyond the application functionality, privacy-preserving techniques are thus an important consideration.



(a) (b) (c) (d)

Figure 1: **Reverse Engineering Attack and Mitigations.** (a) Original image. Object detections are marked with orange bounding boxes. (b) Image reconstruction from $1,000$ SIFT descriptors using our reverse engineering attack. The reconstruction gives sufficient fidelity to preserve detectable semantic information. By (c) reducing the number of features or (d) selective suppression around objects, we can reduce the efficacy of the attack and improve privacy.

In this work, we focus on localization as a fundamental component of augmented reality. Localization relies on visual data assets to make a prediction of the location and pose of the user; in particular most established algorithms rely on local feature descriptors. Since these descriptors contain only derived information, they were long thought to be secure.

Unfortunately, recent literature shows that descriptors can be reverse engineered surprisingly well. We show an example in Figure 1. In general, a reverse engineering attack is the process by which an artificial object is deconstructed to reveal its designs, architecture, code or to extract knowledge from the object [11]. For feature descriptors, a reverse engineering attack attempts to reconstruct the original RGB image that was used to derive the feature descriptors. The fidelity to which the original RGB image can be reconstructed

roughly correlates to the severity of the potential risk to privacy. Prior work [51, 6, 9, 29] has shown that feature descriptors are potentially susceptible to such an attack under a range of conditions and configurations. However, there is limited work on quantitatively analyzing privacy implications as well as evaluating potential defenses against such reverse engineering attacks, which our work will explore.

To scope the problem, we first outline a privacy threat model [8] to contextualize the practicality and data assets available to a descriptor reverse-engineering attack. Using these assets, we show potential reverse engineering attacks and quantify the information leakage to evaluate the privacy implications. We then propose mitigation techniques inspired by best practices in privacy and security [53]. In particular, we propose two mitigation techniques: (1) reducing the number of features shared and (2) selective suppression of features around potentially sensitive objects. We show that these techniques can mitigate the potency of reverse engineering attacks on feature descriptors to improve privacy protections on user data. In summary, we make the following contributions:

1. We present a privacy threat model for a reverse engineering attack to narrow down the privacy-critical information and scope the setup for a practical attack.

2. We demonstrate a reverse engineering attack to reconstruct RGB images from sparse feature descriptors such as FREAK [2], SIFT [22] and SOSNet [45], and quantitatively analyze the privacy implications. In contrast to previous work [29, 9], our approach does not take additional information such as sparse RGB, depth, orientation, or scale as input.

3. We present two mitigation techniques to improve privacy by reducing the number of keypoints shared for localization. We show that there is a trade-off between enhanced privacy (less fidelity of reconstruction) and the utility (localization accuracy). We also show that which keypoints are shared matters for privacy.

2. Related Work

The concept of reverse engineering local features has evolved over recent years as local descriptors play an increasingly important role. Prior work focused primarily on better understanding the image features. Only recently have there been proposals towards leveraging this line of research to understand the privacy implications. Work towards discovering vulnerabilities and mitigating against attacks remains an emerging area of research.

2.1. Recovering Images from Feature Vectors

Reconstruction from Sparse Local Features. Weinzaepfel et al. [51] demonstrated the feasibility of reconstructing the input image, given SIFT [22] descriptors and their keypoint locations, by finding and stitching the nearest neighbors in a database of patches. d'Angelo et al. [6] cast the reconstruction problem as regularized deconvolution problem to recover the image content from binary descriptors, such as FREAK [2] and ORB [35], and their keypoint locations. Kato and Harada [16] showed that it is possible to recover some of the structures of the original image from an aggregation of sparse local descriptors in bag-of-words (BoW) representation, even without keypoint locations. While the quality of reconstructed images from the above methods is far from the original images, they allow clear interpretations of the semantic image content. In this paper, we demonstrate that reverse engineering attacks using CNNs reveal much more image details and quantitatively analyse privacy implications for floating-point [22], binary [2] and machine-learned descriptors [45].

Reconstruction from Dense Feature Maps. Vondrick et al. [49] perform a visualization of HoG [55] features in order to understand its gaps for recognition tasks. To understand what information is captured in CNNs, Mahendran and Vedaldi [23] showed the inversions of CNN feature maps as well as a differentiable version of DenseSIFT [21] and HoG [55] descriptors using gradient descent. Dosovitskiy and Brox [9] took an alternative approach to directly model the inverse of feature extraction for HoG [55], LBP [27] and AlexNet [18] using CNNs, and qualitatively show better reconstruction results than the gradient descent approach [23]. They also show reconstructions from SIFT [22] features using descriptor, keypoint, scale, and orientation information. All the above approaches differ from ours in that we perform the reconstruction from descriptors and keypoints only.

Modern Reverse Engineering Attacks. In the context of 3D point clouds and the AR/VR applications built on top of them, a common formulation of the reverse engineering attack is to synthesize scene views given the 3D reconstruction information. Recent work by Pittaluga et al. [29] showed that it is possible to reconstruct a scene from an arbitrary viewpoint from SfM models using the projected keypoints, sparse RGB values, depth, and descriptors. Our work extends this approach by considering only the modalities available to an attacker as input, which are keypoints and descriptors.

2.2. Defences and Mitigations

Mitigations for Attacks on Sparse Local Features. For reverse engineering attacks on local features, one notable recent work [42, 13, 41] proposes using line-based features to obfuscate the precise location of keypoints in the scene to make the reconstruction difficult. The key idea is to lift every keypoint location to a line with a random direction, but passing through the original 2D [13] or 3D keypoints [42]. Since the feature location can be anywhere on a line, this alleviates privacy implications in the standard mapping

and localization process. Shibuya et al. [41] later extended this approach for SLAM. Similarly, Dusmanu et al. [10] represent a keypoint location as an affine subspace passing through the original point, as well as augmenting the subspace with adversarial feature samples, which makes it more difficult for an adversary to recover original image content.

Mitigations on Raw Images. Apart from local features, other works try to alleviate the privacy concern around sharing raw images by perturbing the images [33, 19, 4, 36, 31, 52, 28, 50]. One way of achieving this is to mask out or replace the parts of images (e.g., faces) that contain private information [48, 33, 19]. Another stream of work focuses on encoding schemes or degrading images to prevent recognition of private image content [4, 36, 31, 52, 28, 50]. A few cryptographic methods were proposed to encrypt visual content in a homomorphic way on local devices [12, 37, 54], which allows computing on encrypted data without decrypting. However, such methods are computationally expensive and it is not clear how to apply them to complex applications such as localization.

2.3. Relationship to Adversarial Attacks on Neural Networks

Recent work has shown that it is possible to trick deep learning models with adversarial inputs to induce incorrect outputs [44, 26, 3, 1]. For example, an adversarial attack may engineer a perceptually indistinguishable input image to trick a deep learning model into emitting an incorrect classification result.

Conceptually, these adversarial attacks are similar to the defense or mitigation strategies that we will propose, since state-of-the-art reverse engineering attacks on descriptors rely on deep learning models. Our mitigation techniques modify inputs in a way to prevent the deep learning model used in the attack from accomplishing its objective — reverse engineering the image. However, unlike prior work in this space, our work lifts the insight that inputs can be modified to induce incorrect outputs and leverages it to **defend** against reverse engineering attacks to preserve privacy instead of as an attack vector.

3. System and Threat Definition

In this section, we first define privacy and utility as well as their trade-offs. We also describe our privacy threat model, which defines assumptions on adversary behavior and the conditions for a practical reverse engineering attack.

3.1. Definitions

Privacy. The LINDDUN privacy threat modeling methodology, one particular methodology in academic discussions, looks at privacy through the following properties [8]: linkability, identifiability, non-repudiation, detectability, information disclosure, content unawareness, and policy. The idea behind LINDDUN is that whenever users share information, one or more of these privacy properties may be at risk. That is relied on for the notion that minimizing the amount of shared information improves privacy. However, precisely quantifying the impact on privacy is application-specific and can be implemented as a continuum, modulating the amount of information to be shared as required. In this work, references to privacy risk and/or threat applies specifically to reidentification risk that comes as a direct result of the reverse engineering attack we describe and evaluate the trade-offs with in Section 5.2)

Utility. Utility captures the accuracy (or performance) of an application or how useful a data asset is to an application. Applications may have multiple utility functions to present a well-rounded understanding of the operation. Utility generally presents a trade-off with privacy as performance tends to improve with dataset size, e.g., ML training. In our case, we use feature matching recall as a proxy for localization accuracy (see Section 5.2).

Privacy-Utility Trade-Off. Applying privacy-preserving techniques can adversely affect utility. The ideal objective of the system is to have both high utility and preserve privacy, but in practice there is a fundamental **trade-off** between the amount of information that one is willing to share and the utility one receives from sharing it. In our case, this means there is a trade-off between the desired localization accuracy (utility) and the images from the user that may potentially be revealed (privacy). The descriptor-based localization service offers a balance between privacy and utility; features sent to the server are still useful to the application pipeline but do not directly leak the rich information content of RGB images which contains private information.

In certain cases where the definitions of utility and privacy are simple, this trade-off can be formalized and reasoned about analytically (e.g. k-anonymity [43]). In larger systems this is not possible and we must **actively** play the roles of attacker and defender to model possible attacks and understand the risks to user privacy. In computer security and privacy, this is the role of a *privacy threat model* [39, 25, 46, 47, 24, 38, 8].

3.2. Privacy Threat Model

Building a privacy threat model is application specific. For our localization use-case, the closest is the LINDDUN "hard privacy" threat model [8] where the objective is to share as little information as possible to a potential adversary. At a high level, LINDDUN proposes building a dataflow diagram of a system, data assets, adversary, and potential attack vectors. These are then used to audit potential threats that may impact privacy properties. In our work, we focus on identifiability, detectability, and information disclosure, which are the most relevant to our reverse engineering attack on RGB images. *Identifiability* refers to whether an adver-

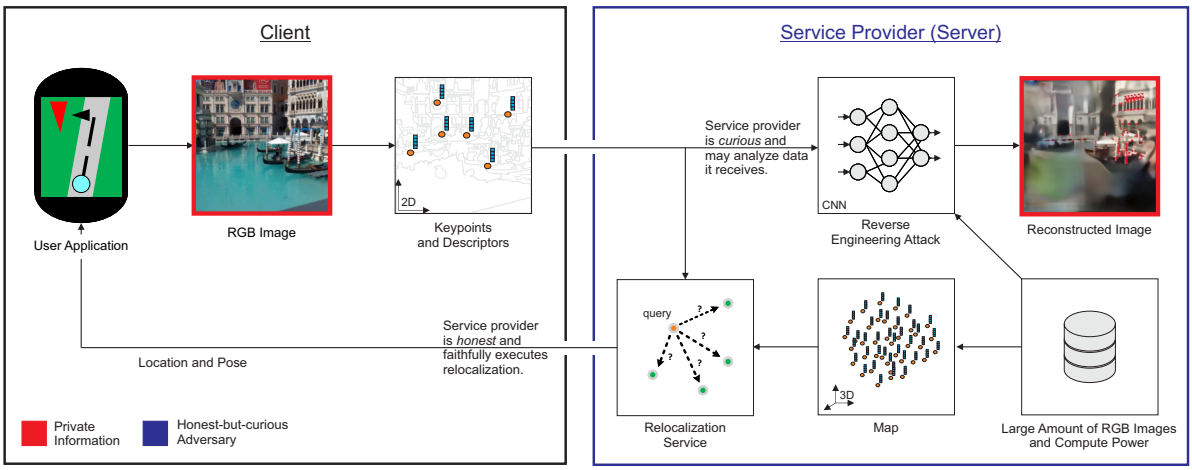


Figure 2: **Overview of a typical localization system and our privacy threat model.** A client derives descriptors from RGB images and shares only the descriptors with a service provider. The service provider is honest and faithfully executes relocalization by matching query descriptors against those of a map. The service provider is also curious and may attempt to derive insight about the user. This sets up a potential reverse engineering attack. For such a system, our mitigation strategy must lie at the interface between the client device and the honest-but-curious service provider. Minimizing information shared at this interface reduces the amount of sensitive information that reaches the adversary.

sary can identify items of interest. *Detectability* refers to whether an adversary can detect whether items exist or not. *Information disclosure* refers to whether information about the user is disclosed to an adversary who should not have access to it. An adversary with an RGB image can observe information about each of these properties which poses a risk to privacy.

System Definition and Sensitive Data Assets. Figure 2 shows the relevant components of our privacy threat model. Our system follows a client-server architecture to process localization requests. For localization, there are two primary data assets: (1) RGB images and (2) feature descriptors. The client takes RGB images and derives feature descriptors which are shared with the server to query the user's location and pose from a map. We focus on protecting the RGB images as these are data assets which could be used to identify items of interest. Descriptors are perceived as more private and more acceptable to share because they do not **directly** leak RGB information. However in Section 5.3, we will show that indirectly this is not true.

Adversary Definition and Potential Attacks. Our privacy threat model considers the service provider as an adversary (Figure 2) that is *honest-but-curious*, which is canonical in the security literature [15]. The honest-but-curious adversary is a legitimate participant in the system and executes the agreed upon application or service faithfully (as opposed to outright malicious behavior). But, while fulfilling the service, the adversary is *curious* and may use available data to learn information about the client. In our case, the adversary poses a risk to the client's privacy by reverse engineering the RGB images from feature descriptors. This is possible because the adversary has access to similar data – specifically feature descriptors and source RGB images – and large scale compute resources. Together, this means an adversary is capable of training deep-learning models (such as a reverse engineering model) to analyze data in a reasonable amount of time.

The goal of this paper is to understand how a client's protection against an honest-but-curious adversary capable of training deep learning models to reverse engineer RGB images from feature descriptors could be enhanced.

4. Reverse Engineering Attack

This section defines the convolutional neural network models we use to craft our reverse engineering attack. As shown in Figure 2, this model takes sparse local features (keypoints and descriptors) as input and estimates the original RGB image.

4.1. Model Architecture

Given a user image $\mathbf{I}(i,j) \in \mathbb{R}^3$ and a derived sparse feature map $\mathbf{F}_{\mathbf{I},M}(i,j) \in \mathbb{R}^C$ containing C-dimensional local descriptors from the image $\mathbf{I}$ using a feature extractor M, we seek to reconstruct an image $\hat{\mathbf{I}}(i,j) \in \mathbb{R}^3$ from $\mathbf{F}_{\mathbf{I},M}$. The sparse feature map is assembled by starting with zero vectors and placing extracted descriptors at keypoint locations i,j. Our reverse engineering attack relies on a deep convolutional generator-discriminator architecture that is trained for each specific feature extraction method M. The generator G_M produces the reconstructed image:

$$\hat{\mathbf{I}} = G_M(\mathbf{F}_{\mathbf{I},M})$$

and follows a single 2-dimensional U-Net topology [34] with 5 encoding and 5 decoding layers as well as skip connections with convolutions. The discriminator D_M is a 6 layer convolutional network operating on top of G_M [30]. Please see the supplemental material for details. In order to adhere to our privacy threat model and in contrast to prior work by Pittaluga et al. [29], we do not use depth or RGB inputs and subsequently also do not make use of a VisibNet.

4.2. Loss Functions

We use the following loss functions to train the reconstruction network:

MAE. The mean absolute error (MAE) is the pixelwise L1 distance between the reconstructed and ground truth RGB images:

$$L_{mae} = \sum_{i,j} ||\hat{\mathbf{I}}(i,j) - \mathbf{I}(i,j)||_1 \,. \tag{1}$$

L2 Perceptual Loss. The L2 perceptual loss is measured as:

$$L_{perc} = \sum_{i,j} \sum_{k=1}^{3} ||\phi_k(\hat{\mathbf{I}}(i,j)) - \phi_k(\mathbf{I}(i,j))||_2^2 \,, \tag{2}$$

with ϕ_k being the outputs of a pre-trained and fixed VGG16 ImageNet model [7]. ϕ_k are taken after the ReLU layer k with $k \in \{2, 9, 16\}$.

BCE. For the generator-discriminator combination, we use the binary cross-entropy (BCE) loss defined as:

$$L_{bce} = \sum_{i,j} log(D_M(\hat{\mathbf{I}}(i,j))) + log(1 - D_M(\mathbf{I}(i,j))) \,. \tag{3}$$

Finally, we optimize the losses together:

$$L_G = L_{mae} + \alpha L_{perc} + \beta L_{bce} \,, \tag{4}$$

with α and β as scaling factors.

5. Evaluation

5.1. Experimental Setup

Sparse Local Features. For the feature extraction method M from Section 4.1, we use SIFT [22] ($C = 128$), FREAK [2] ($C = 64$), and SOSNet [45] descriptors ($C = 128$) as representatives of traditional and machine-learned variants. Keypoint locations for FREAK and SOSNet were detected using Harris corner detection [14]. For reconstruction, we use the SIFT detector for SIFT descriptors as in [29]; however, for image matching we use Harris corners for SIFT descriptors because we found the SIFT detector performed poorly in this setting.

Training and Evaluation Data. We train our networks on $50,000$ images and their extracted sparse local features from the training partition of the MegaDepth dataset [20]. For testing the reverse engineering attack, we sampled $9,800$ images from the MegaDepth test set that contain objects as candidates for potential private data.

Network Training. A different reverse engineering model M is trained for 400 epochs for each descriptor type. The learning rate is initialized to 0.001 and 0.0001 for the generator and discriminator networks respectively. Learning rates are adjusted using the Adam optimizer [17].

5.2. Measuring Privacy and Utility

Measuring Privacy with SSIM. Our first metric for measuring privacy is structural similarity (SSIM), which measures the perceptual similarity between images. In our case, we use SSIM to evaluate how much visual information the reverse engineering attack can recover by comparing against the original image. Therefore, SSIM provides a way to measure identifiability. We note that the SSIM measures to what extent the **whole** image may be recovered, which includes private and public information (e.g. people and buildings respectively); the public information is also available to the service provider when building the map. However, measuring how well the whole image can be reconstructed includes the reconstruction quality of private regions. SSIM can further serve as a proxy to estimate how well other tasks such as object detection, landmark recognition, and optical character recognition may perform on the reverse-engineered image.

Measuring Privacy by Object Detection. We use an object detector (YOLO v3 [32], with 80 classes) to measure how much semantic information can be inferred from the reverse-engineered images. We compare object detection results on both the original and the reconstructed images. If an object's bounding box in the original image has at least 50% overlap with that of the reconstructed image of the same class label, we consider them as a match. The more correspondence between objects in the original and the reconstructed image, the higher the risk to privacy.

Measuring Utility. To assess utility of local features when applying our mitigation strategies, we define an *image matching* task as a proxy for localization and investigate how the feature matching between two images deteriorates as we increase the privacy. Specifically, we generate corresponding image pairs from the 53 landmarks of the test split of the MegaDepth [20] dataset. For each landmark, we sample 50 pairs of images that have at least 20 covisible 3D points determined from a reference map built with COLMAP [40], resulting in $2,650$ image pairs. For each corresponding pair of images, we perform local correspondence matching using input features, and count the number of pairs with at least 20 inlier matches which we deem as successful. We refer to the proportion of image pairs that have been successfully

Figure 3: **Reverse Engineering Attack Results.** From top to bottom: ground truth and reconstructions from a max. of $1,000$ sparse SIFT, FREAK and SOSNet features. One can observe that reconstruction from only sparse local features reveals the original image information extremely well. Note that the images show landmarks that were not included in the training data.

matched as our matching recall, which we use as our utility measure.

5.3. Reverse Engineering Attack

We first evaluate to what extent the reverse-engineering attack from Section 4 poses a risk to privacy. Examples of the reconstructions are shown in Figure 3 and the privacy metrics of the reverse-engineered images are given in Table 1. Reconstructions using FREAK [2] descriptors yield substantially poorer reconstruction quality and semantic content than SIFT [22] and SOSNet [45]. Despite differences in feature extraction techniques and descriptor sizes, all three descriptors are susceptible to the attack and yield reconstructions comparable to prior work [29] (please see supplemental material for detailed comparison to prior work), but notably without RGB or depth information as input. At a higher level, the results show that the reverse engineering attack can introduce a reidentification risk of RGB image content. The results from Table 1 also show that the reverse-engineered images still allow an adversary to detect and identify some objects that were present in the original images.

Descriptor	SSIM	Detected Objects
SIFT [22]	0.675	32.58%
FREAK [2]	**0.511**	**19.32%**
SOSNet [45]	0.616	41.26%

Table 1: **Privacy metrics of reverse-engineered images using $1,000$ keypoints.** The amount of detected objects using YOLO v3 [32] is measured on the reverse-engineered images relative the amount detected on the original images. FREAK descriptors reveal less information than SIFT and SOSNet.

5.4. Mitigation by Reduction of Features

Following Section 3.2, to improve privacy, our objective is to minimize the information shared by the client. To this end, we investigate how reducing the number of features increases privacy at the expense of utility.

For each descriptor type, we retain a maximum of N top-scoring keypoints based on the detector response and vary N from 1000 to 100. For each value of N we then evaluate how well our reverse-engineering models perform. Qualitative results are given in Figure 4. We show the average privacy (measured by $1-$SSIM) of the reconstructed images vs. the number of features in Figure 5a. The data shows the degradation in SSIM of the reconstructed images accelerates

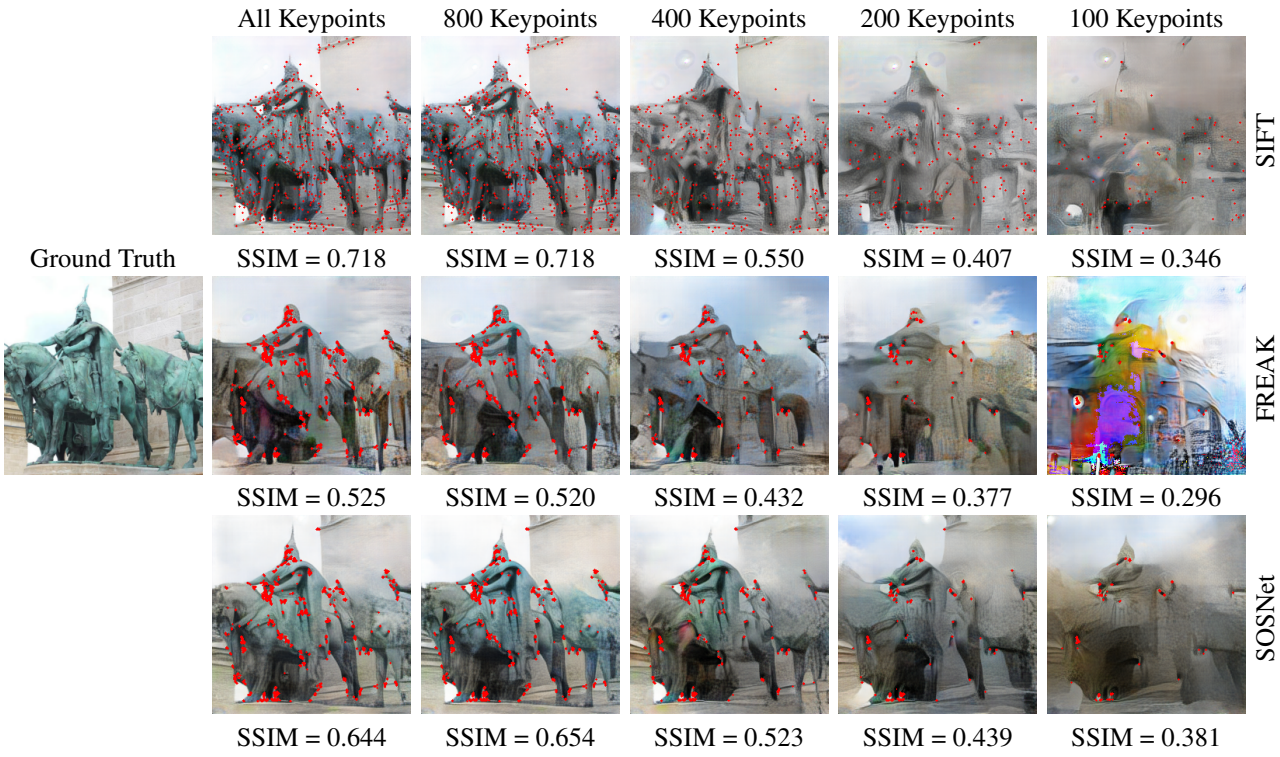
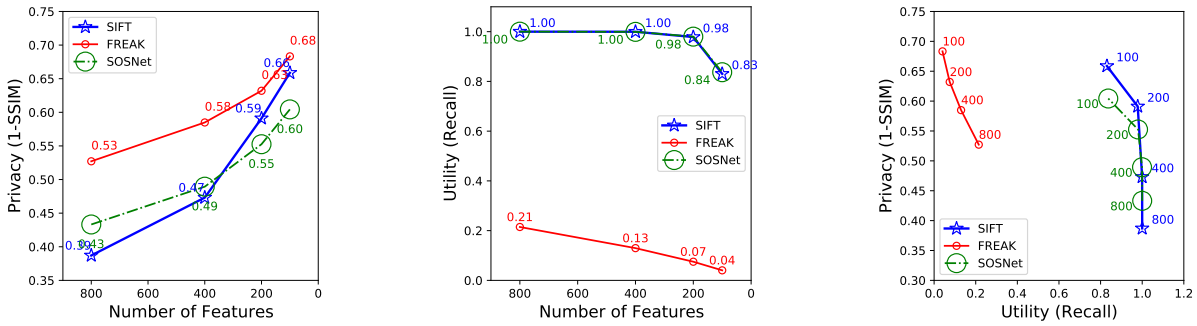


Figure 4: **Reverse engineering ablation study of reducing keypoints.** SIFT, FREAK and SOSNet reverse engineering results using $1,000$, 800, 400, 200, and 100 keypoints respectively, annotated in red. Reducing keypoints reduces the potency of the reverse engineering attack. Regions with higher densities of keypoints have better reconstruction quality.



(a) (b) (c)

Figure 5: **Utility and Privacy Trade-Off when Varying the Number of Features.** Privacy increases when reducing the number of features where FREAK gives the best results. For utility, FREAK and SIFT gives the best results. SIFT gives the best overall trade-off.

as more keypoints are removed. For less than 300 features, SIFT gives better results than SOSNet. FREAK outperforms SIFT and SOSNet, and yields the best results in terms of privacy.

However, despite strong privacy results, FREAK trades-off utility. In Figure 5b, we show how the utility changes. Here, FREAK gives the lowest utility, indicating that FREAK descriptors overall provide less useful information than SOSNet and SIFT. Interestingly, for SOSNet and SIFT the number of keypoints can be reduced to 200 by sacrificing only 2% performance. The trade-off between utility and privacy is shown in Figure 5c. Overall, we find that SIFT yields the best privacy-utility trade-off among the evaluated descriptor configurations on the Megadepth dataset. We note that these results do not preclude the possibility that other descriptor configurations (i.e., in terms of dimensionality, target dataset, and type) may achieve better results. Ultimately the ideal descriptor chosen will depend on the precise privacy and utility requirements necessitated by the localization service.

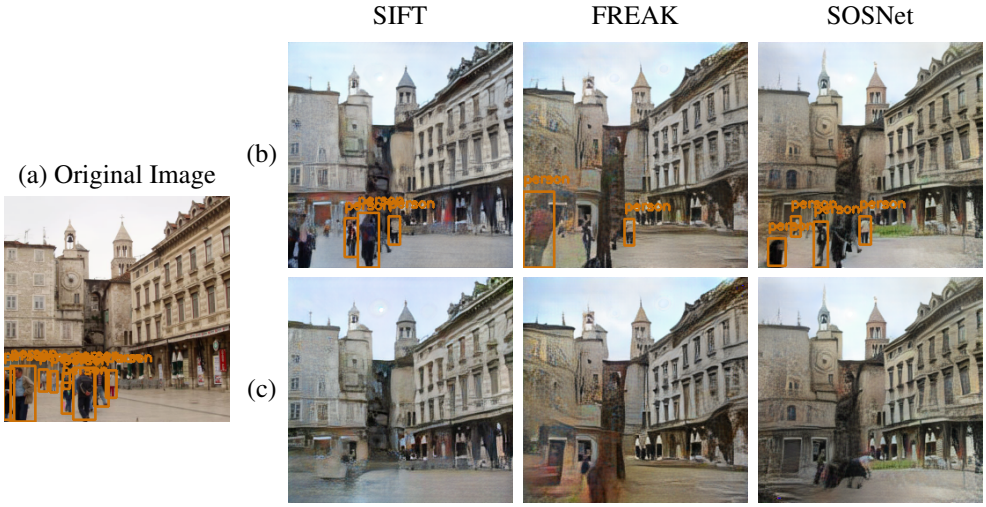


Figure 6: **Reverse Engineering Results after Selective Feature Supression.** (a) Original image with object detection results (people). (b) Reverse-engineered images without feature suppression (with max. number of keypoints set to 1000), followed by object detection. (c) Reverse-engineered images using feature suppression, followed by object detection. As visible, all objects that can be detected by the YOLOv3 object detector without the suppression are successfully removed with the suppression.

5.5. Selective Suppression of Features

Globally reducing image features can reduce the potency of the reconstruction attack, but at the same time it reduces the matching accuracy. In this section, we investigate to what extent an object detector can help implement a more selective approach. We identify and mark the sensitive regions in the images using the bounding boxes produced by the YOLO v3 [32] object detector. Based on the bounding boxes, we then suppress any features in these regions. Finally, we apply our reverse-engineering attack and measure the detectable semantic information content in the images before and after reverse engineering (Table 2).

Figure 6 shows a qualitative example of how selective feature suppression effectively defeats the object detector; the people detected in the original image do not appear nor are identifiable by the object detector in the reconstructed images. These results confirm our intuition that selective suppression can effectively preserve the privacy around a potentially sensitive region of interest (in our case semantic content of people in the image). Note that the quality of the overall image outside of the marked sensitive regions remains largely unaffected. Finally, the results show that features of private objects should not be shared in order to mitigate privacy risks posed by reverse engineering attacks.

Results for the privacy-utility trade-off of the suppression are given in Table 2. Under the evaluated experimental conditions, SIFT and SOSNet give better trade-offs than FREAK; these trends are consistent with the results from Section 5.4. Notably for SIFT the utility drops slightly, while the detected objects are almost eliminated.

	Privacy (Object Recall)		Utility (Matching Recall)	
Supression	No	Yes	No	Yes
SIFT [22]	20%	**2.21%**	100%	**88%**
FREAK [2]	11%	1.29%	34%	28%
SOSNet [45]	28%	5.21%	100%	88%

Table 2: **Privacy-Utility Trade-Off for Selective Feature Suppression.** Object recall shows how many objects can be detected from the reverse engineered images compared to the original images without and with suppression (note that lower is better). Matching recall shows how many images can be successfully matched without and with selective feature suppression. SIFT gives the best overall trade-off.

6. Conclusion

Our work has formulated a privacy threat model to scope the threats to descriptor-based localization. In contrast to prior work, for the first time, we have shown a reverse engineering attack that operates in the real-world scenario, where only sparse local features are available to an honest-but-curious adversary. We found that our reverse engineering attack could reconstruct the original image with surprisingly good quality. We then investigated two mitigation techniques and showed a trade-off between privacy and utility (measured by feature matching). We found that using an object detector to suppress objects slightly reduces matching accuracy (as a proxy for localization accuracy) but gives better privacy results (fewer reidentifiable objects). Finally, our analysis has shown that, among the descriptors and we evalu-

ate, the best overall privacy-utility trade-off can be achieved with SIFT, when compared to FREAK and SOSNet. Privacy (defined as reidentification risk through reverse engineering attacks as specifically described in this paper) may be preserved with the mitigation techniques described in this paper. Looking forward, our work provides initial experiments on some mitigation techniques the community may consider to further the privacy-aware descriptor-based applications research.

References

[1] N. Akhtar and A. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018. 3

[2] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst. Freak: Fast retina keypoint. In *CVPR*, 2012. 2, 5, 6, 8

[3] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013. 3

[4] Daniel J Butler, Justin Huang, Franziska Roesner, and Maya Cakmak. The privacy-utility tradeoff for remotely teleoperated robots. In *HRI*, 2015. 3

[5] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the cloud. *Acm Sigact News*, 40(2):81–86, 2009. 1

[6] Emmanuel d'Angelo, Laurent Jacques, Alexandre Alahi, and Pierre Vandergheynst. From bits to images: Inversion of local binary descriptors. *TPAMI*, 36(5):874–887, 2013. 2

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 5

[8] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16(1):3–32, 2011. 2, 3

[9] Alexey Dosovitskiy and Thomas Brox. Inverting visual representations with convolutional networks. In *CVPR*, 2016. 2

[10] Mihai Dusmanu, Johannes L Schönberger, Sudipta N Sinha, and Marc Pollefeys. Privacy-preserving visual feature descriptors through adversarial affine subspace embedding. *arXiv preprint arXiv:2006.06634*, 2020. 3

[11] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley Sons, Inc., USA, 2005. 1

[12] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *International symposium on privacy enhancing technologies symposium*, 2009. 3

[13] Marcel Geppert, Viktor Larsson, Pablo Speciale, Johannes L Schönberger, and Marc Pollefeys. Privacy preserving structure-from-motion. ECCV, 2020. 2

[14] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988. 5

[15] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *USENIX*, 2018. 4

[16] Hiroharu Kato and Tatsuya Harada. Image reconstruction from bag-of-visual-words. In *CVPR*, 2014. 2

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 5

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 2017. 2

[19] Tao Li and Lei Lin. Anonymousnet: Natural face de-identification with measurable privacy. In *CVPRW*, 2019. 3

[20] Zhengqi Li and Noah Snavely. Megadepth: Learning single-view depth prediction from internet photos. In *CVPR*, 2018. 5

[21] Ce Liu, Jenny Yuen, and Antonio Torralba. Sift flow: Dense correspondence across scenes and its applications. *TPAMI*, 2010. 2

[22] David G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, 1999. 2, 5, 6, 8

[23] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *CVPR*, 2015. 2

[24] MM Morana. Wiley: Risk centric threat modeling: Process for attack simulation and threat analysis-tony ucedavelez, marco m. morana. accessed on 09/05/2016. 3

[25] Suvda Myagmar, Adam J Lee, and William Yurcik. Threat modeling as a basis for security requirements. In *Symposium on requirements engineering for information security (SREIS)*, volume 2005, pages 1–8. Citeseer, 2005. 3

[26] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *CVPR*, 2015. 3

[27] Timo Ojala, Matti Pietikainen, and Topi Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *TPAMI*, 2002. 2

[28] Francesco Pittaluga, Sanjeev Koppal, and Ayan Chakrabarti. Learning privacy preserving encodings through adversarial training. In *WACV*, 2019. 3

[29] Francesco Pittaluga, Sanjeev J Koppal, Sing Bing Kang, and Sudipta N Sinha. Revealing scenes by inverting structure from motion reconstructions. In *CVPR*, 2019. 2, 5, 6

[30] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. 5

[31] Nisarg Raval, Ashwin Machanavajjhala, and Landon P Cox. Protecting visual secrets using adversarial nets. In *CVPRW*, 2017. 3

[32] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018. 5, 6, 8

[33] Zhongzheng Ren, Yong Jae Lee, and Michael S Ryoo. Learning to anonymize faces for privacy preserving action detection. In *CVPR*, 2018. 3

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI*. Springer, 2015. 5

[35] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *ICCV*, 2011. 2

[36] Michael S Ryoo, Brandon Rothrock, Charles Fleming, and Hyun Jong Yang. Privacy-preserving human activity recognition from extreme low resolution. *arXiv preprint arXiv:1604.03196*, 2016. 3

[37] Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *International Conference on Information Security and Cryptology*, 2009. 3

[38] Paul Saitta, Brenda Larcom, and Michael Eddington. Trike v. 1 methodology document [draft]. *URL: http://dymaxion. org/trike/Trike v1 Methodology Documentdraft. pdf*, 2005. 3

[39] Chris Salter, O Sami Saydjari, Bruce Schneier, and Jim Wallner. Toward a secure system engineering methodolgy. In *Proceedings of the 1998 workshop on New security paradigms*, pages 2–10, 1998. 3

[40] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *CVPR*, 2016. 5

[41] Mikiya Shibuya, Shinya Sumikura, and Ken Sakurada. Privacy preserving visual slam. *arXiv preprint arXiv:2007.10361*, 2020. 2, 3

[42] Pablo Speciale, Johannes L Schonberger, Sing Bing Kang, Sudipta N Sinha, and Marc Pollefeys. Privacy preserving image-based localization. In *CVPR*, 2019. 2

[43] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002. 3

[44] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013. 3

[45] Yurun Tian, Xin Yu, Bin Fan, Fuchao Wu, Huub Heijnen, and Vassileios Balntas. Sosnet: Second order similarity regularization for local descriptor learning. In *CVPR*, 2019. 2, 5, 6, 8

[46] Peter Torr. Demystifying the threat modeling process. *IEEE Security & Privacy*, 3(5):66–70, 2005. 3

[47] Tony UcedaVelez. Real world threat modeling using the pasta methodology. *OWASP App Sec EU*, 2012. 3

[48] Nishant Vishwamitra, Bart Knijnenburg, Hongxin Hu, Yifang P Kelly Caine, et al. Blur vs. block: Investigating the effectiveness of privacy-enhancing obfuscation for images. In *CVPRW*, 2017. 3

[49] Carl Vondrick, Aditya Khosla, Tomasz Malisiewicz, and Antonio Torralba. Hoggles: Visualizing object detection features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1–8, 2013. 2

[50] Zihao W. Wang, Vibhav Vineet, Francesco Pittaluga, Sudipta N. Sinha, Oliver Cossairt, and Sing Bing Kang. Privacy-preserving action recognition using coded aperture videos. In *CVPRW*, 2019. 3

[51] Philippe Weinzaepfel, Hervé Jégou, and Patrick Pérez. Reconstructing an image from its local descriptors. In *CVPR*, 2011. 2

[52] Zhenyu Wu, Zhangyang Wang, Zhaowen Wang, and Hailin Jin. Towards privacy-preserving visual recognition via adversarial training: A pilot study. In *ECCV*, 2018. 3

[53] Kim Wuyts and Wouter Joosen. Linddun privacy threat modeling: a tutorial. *CW Reports*, 2015. 2

[54] Ryo Yonetani, Vishnu Naresh Boddeti, Kris M Kitani, and Yoichi Sato. Privacy-preserving visual learning using doubly permuted homomorphic encryption. In *ICCV*, 2017. 3

[55] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *CVPR*, 2006. 2