Camouflage: Utility-Aware Obfuscation for Accurate Simulation of Sensitive Program Traces

ASMITA PAL, University of Wisconsin-Madison,, USA
KEERTHANA DESAI, Google, Mountain View, USA
RAHUL CHATTERJEE, University of Wisconsin-Madison, Madison, USA
JOSHUA SAN MIGUEL, University of Wisconsin-Madison, Madison, USA

Trace-based simulation is a widely used methodology for system design exploration. It relies on realistic traces that represent a range of behaviors necessary to be evaluated, containing a lot of information about the application, its inputs and the underlying system on which it was generated. Consequently, generating traces from real-world executions risks leakage of sensitive information. To prevent this, traces can be obfuscated before release. However, this can undermine their ideal utility, i.e., how realistically a program behavior was captured. To address this, we propose Camouflage, a novel obfuscation framework, designed with awareness of the necessary architectural properties required to preserve *trace utility*, while ensuring secrecy of the inputs used to generate the trace. Focusing on memory access traces, our extensive evaluation on various benchmarks shows that camouflaged traces preserve the performance measurements of the original execution, with an average τ correlation of 0.66. We model input secrecy as an input indistinguishability problem and show that the average security loss is 7.8%, which is better than traces generated from the state-of-the-art.

CCS Concepts: • **Computer systems organization → Architectures**; • **Security and privacy → Security in hardware**; **Information flow control;**

Additional Key Words and Phrases: Synthetic trace generation, Privacy of traces, Performance characterization

ACM Reference Format:
Asmita Pal, Keerthana Desai, Rahul Chatterjee, and Joshua San Miguel. 2024. Camouflage: Utility-Aware Obfuscation for Accurate Simulation of Sensitive Program Traces. *ACM Trans. Arch. Code Optim.* 21, 2, Article 36 (May 2024), 23 pages. https://doi.org/10.1145/3650110

1 INTRODUCTION

The computer architecture community has invested significant effort in developing modeling and simulation tools for running standard algorithms and benchmark suites. Although such benchmark suites span a wide expanse of architectural behaviors, with the increasing rate of application

K. Desai work was completed while at University of Wisconsin-Madison.
Authors' addresses: A. Pal and J. San Miguel, University of Wisconsin-Madison, 1415 Engineering Drive, Madison, WI, USA, 53705; e-mails: asmita.pal@wisc.edu, jsanmiguel@wisc.edu; K. Desai, Google, 1600 Amphitheatre Parkway, Mountain View, CA, USA, 94043; e-mail: keerthanadesai@google.com; R. Chatterjee, University of Wisconsin-Madison, 1210 W Dayton St, Madison, WI, USA, 53705; e-mail: rchatterjee4@wisc.edu.

ACM 1544-3973/2024/05-ART36
https://doi.org/10.1145/3650110

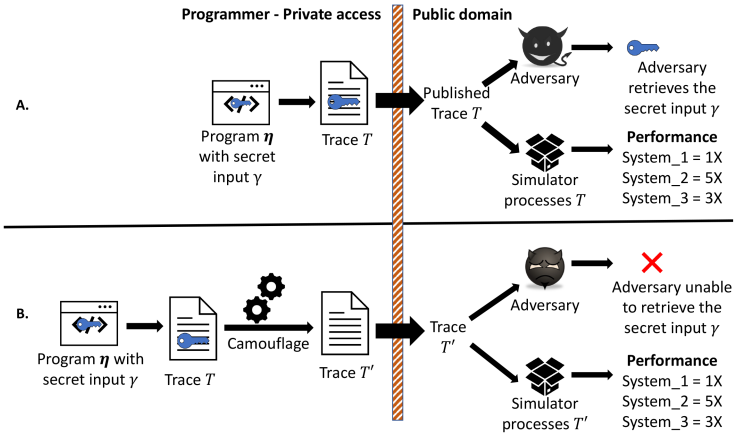


Fig. 1. Utility and secrecy risk of publishing traces. c attempts to preserve the utility, while protecting secrecy of input.

development and algorithmic complexity, it becomes challenging for standard benchmarks to capture the diverse workloads emerging everyday [46]. Thus, there has been a growing trend towards the release and dissemination of *traces* that are profiled from real-world executions of important applications. Such traces allow the research community to conduct hardware design space exploration with more realistic and timely application behavior. For example, Google recently released a set of warehouse-scale computing workload traces [37] to provide a better connection between the evaluation benchmarks used by academic researchers and the kinds of programs actually run on Google servers on a regular basis. In fact, each year, competitions are held in computer architecture conferences using trace-based simulation to determine the best performing prefetchers [5], branch predictors [1], and cache replacement policies [4].

The challenge is that, as we work towards optimizing for real-world applications, we often inadvertently capture sensitive information about program input data, while modelling program behavior. For example, hyperparameters and training data used for training a machine learning model to predict Gmail smart compose [17] may contain sensitive user inputs. An adversary can analyze such a published trace and recover information about the secret input (γ, as shown in Figure 1(a). On the contrary, real inputs are tied to an application's memory footprint and workflow, which in turn are important for characterizing system performance. Capturing these accurately is critical to the *utility* of a trace (simulator performance measurements in Figure 1). Prior techniques for synthetic trace generation have focused mainly on utility and minimizing simulation speed [7, 8, 10, 47, 58, 59]. They do not consider potential leakage of information about the input in the trace. The challenges of figuring out an optimal trade-off for *utility* and *security* have been explored in the context of program traces [18, 19]. Synthetic traces generated from these techniques limit the amount of information being passed onto the trace. However, this information bound is user-defined; a less conservative choice by the programmer could lead to privacy exploits, while a more conservative choice will obfuscate necessary architectural behavior, thus jeopardizing the *trace utility*. Another instance of *trace security* has been studied by prior works on memory trace obliviousness, which leverage a similar problem of program input data being secret and the program code not being secret [39]. In such a threat model, secret data influence the execution path, thus changing the content and length of memory access, thus making it prone to adversarial exploits.

Motivation Generalizing *utility* and *security* measures across traces from all applications continues to be a hard problem. For security, traces such as those of AES encryption can easily be gen-

erated using a trivial secret key that is not related to the actual key. But such traces are small and generally not adopted for performance characterization. Utility-oriented traces are more reliant on input data, which are often large in size. First, it is hard to quantify the privacy of such inputs which do not constitute a secret key. Second, generating synthetic inputs that can mimic the architectural behavior of real inputs is a challenge in itself (as we describe in Section 5.4). Even the usage of machine learning models on such data requires several samples of the input, which are often not readily available. Existing approaches impose either too great a burden on the party that wants to share the trace or undermines the utility of the trace, since it is challenging to obtain both (1) *utility* — retain features that are important for architectural optimization, and (2) *security* — hide the parts of the trace that may be sensitive. We see that most techniques are only optimized for either utility or security, not both, as our evaluations will show. To bridge this gap, we propose Camouflage, which preserves secrecy (Section 6) of sensitive inputs while delivering optimum utility (Section 5).

Camouflage leverages inherent architectural properties of program semantics to guide behavior of a trace under simulations. These can be inferred either via hints from the programmer or via automated heuristics during trace generation [26, 27, 54]. Camouflage operates in a conservative *security-by-default* manner, prioritizing obfuscation of the trace and making sure it is only compromised when the designer seeks a greater utility trade-off. Such obfuscation ensures immunity against adversaries vying to extract information about application inputs. For example, an attacker can recover the secret key [44] from the memory access trace of an encryption algorithm. However, if the trace is passed through Camouflage as shown in Figure 1, it can obfuscate the secret input γ, making it nearly impossible for the adversary to recover it, for the same class of attacks.

For trace utility, we limit Camouflage's focus on properties influencing cache behavior. This can be attributed to the fact that caches capture complex memory access patterns and our use-case for this study is memory traces. Prefetchers and replacement policies are two key architectural optimizations that are well-studied today. They rely on accurate simulation of the cache and memory hierarchy to capture regularities in memory access streams and leverage some form of spatio-temporal locality. We show through our experimentation, Camouflage is able to preserve the architectural properties of traces relevant for trace-based optimizations, while removing the information about the input from the trace. Camouflage does this without requiring the practitioners to regenerate traces on fake inputs which is cumbersome, or use statistical generators to create traces that do not represent the actual execution of an application.

Contributions. We make the following contributions:

- We identify inherent architectural properties of some key data structures that influence the performance of various cache-related optimization processes.
- We design and build a trace obfuscation system, called Camouflage, with a conservative security-preserving mindset that uses tailored obfuscation techniques based on program semantics. The data structure identification is guided by programmer-led hints or by the structure learned from the trace using a machine learning model.
- Through comprehensive experimentation with various workloads and cache replacement and prefetching policies proposed in prior works, we show that Camouflage-obfuscated traces maintain performance similarity with a τ correlation of 0.66. Additionally, we show that camouflaged trace preserves the security of inputs, with security loss of average 7.8%, over all benchmarks.

2 SECURITY-UTILITY TRADEOFFS IN MEMORY TRACES

Memory access traces are very useful for analyzing a range of architectural behaviors. However, they can also reveal information about the input, which can be sensitive in certain contexts. Preserving the security of the input is paramount to allow free sharing of traces generated on sensitive

inputs. We aim to obfuscate traces such that (1) traces are secure-by-default, prioritizing the mitigation of sensitive input information; and (2) architectural properties are preserved in the trace. In this section, we explore challenges associated with this two-pronged problem of maintaining *security* and *utility*, in the context of memory access traces.

2.1 Trace Structure Notation

Memory traces are often used to characterize program behavior for studying prefetching and replacement policies [1, 4, 5]. In this work, we adopt the trace structure as a sequence of read (load) and write (store) operations along with memory address values. We formally define a memory trace T, as a set of memory accesses by program η on an input γ (Figure 1). We denote the trace generation process f as, $T \leftarrow f(\eta, \gamma, E)$, where E denotes the environment where the program is executed and captures the sources of non-determinism due to scheduling, concurrency, and the like. T is an ordered list of tuples, $T = \{(a_1, v_1, s_1), \ldots, (a_n, v_n, s_n)\}$, where $a_i \in \{\text{load}, \text{store}\}$ is the load or store operation, v_i is a virtual memory address, and s_i represent the size of the (load or store) access, which is typically between 1-8 bytes depending on the data type. Our goal is to ensure that no information of the input γ can be inferred from the trace T.

2.2 Threat Model

Publicly available traces are used for performance optimization. We therefore assume the attacker has access to the trace (T) generated on a secret input (γ) by the company. We also conservatively assume that the attacker has access to the code of the program (η) used to generate the trace. Although secrecy of the program is important in certain context, such as intellectual property of companies, for this paper, we focus on the secrecy of the inputs, which can take various forms like hyper-parameters of an ML model, structure of a user interaction graph, or keys used for an encryption mechanism. Assuming the adversary can obtain the memory access trace and the program, their goal is then to extract some non-trivial information about the input (which they cannot obtain otherwise). It is tricky to define "some non-trivial information". In cryptography research, therefore, it is captured using a (in)distinguishability framework [24], where, the attacker gets to pick two inputs of their choice, and defender (which is the trace generator in our case) returns (obfuscated) trace of one of the inputs chosen at random; the attacker needs to tell which input was chosen by the defender. Now, if the attacker cannot tell which input it was, just by looking at the trace, we can safely assume that it will be hard to learn any non-trivial information about any of the inputs. We will use this stronger notion of security in the paper, instead of entropy-based metrics used in some previous work [18] — indeed even a single bit of security loss can leak sensitive information about the input. Instead, we compute the probability of an adversary to distinguish the inputs, and thereby bound their ability to learn any sensitive information from a given trace. Looking ahead, in Section 6 we establish a classification task that upper bounds attacker's ability to learn any non-trivial information from a trace.

2.3 Exploitable Information in Traces

Given our trace structure, it is important to understand what information about the underlying input can be given out by memory access traces alone. Often memory patterns and usage are a function of application inputs. For example, in Advanced Encryption Standard (AES) encryption, accesses to the s-box table in memory is determined by the Exclusive OR operation (XOR) of the secret key and plaintext [44]. Here, we consider some broad scenarios of potential information leakage from such traces: (a) absolute address values, (b) temporal memory access patterns, and (c) spatial memory access patterns.

Absolute address values. Traces record memory addresses accessed for reading (load) and writing (store) by a program. Such addresses might reveal a secret key or certain information about the system where the trace was generated. Thus, the absolute values must be masked.

Temporal patterns. Temporal patterns arise if a particular memory location (or a group of memory locations) is accessed frequently in a fixed interval. Such patterns occur naturally due to loops or recursion calls. The periodic access can inadvertently act as a signpost to help attackers identify the locations of interest in the trace. More importantly, it may also reveal loop constants, which can be security-critical if they are dependent on the sensitive inputs. For example, if a program compares two strings (such as passwords), looping over the two strings can reveal the length of the strings and how many characters match. Timing side channels (c.f., [15]) extensively exploit such vulnerabilities to extract sensitive information.

Spatial patterns. Relative spatial positions of memory addresses in the trace can reveal the underlying data structure. If the memory accesses are dependent on the sensitive input, relative positions could reveal the sensitive input, as we discuss in Section 6.2 how AES secret key is recovered from a trace. The S-box array is accessed based on the XOR of the secret key and the plaintext, and the relative memory accesses then reveal the output of that XOR. Thus, if the adversary knows some part of the plaintext (e.g., HTML header), they could learn the secret key. Similarly, arrays that record items, based on their hash values, could reveal information about the values that are stored in it. Especially if the values inserted are from a small domain, such as SSN numbers, Zipcodes, or names of medicines. Consequently, an attacker can brute-force to recover the inputs by observing the relative memory accesses of an data-dependent array in a trace. For data-independent arrays, spatial patterns can reveal the size of the array, the ordering of the elements in the array, and even the elements. Prior work in attacking encrypted databases has shown it is possible to recover the elements from just the order of the elements [25].

Known security threats in traces. Research exploiting hardware side channels such as power consumption traces [22] and cache timing information [44], shows that its possible to retrieve secret keys, after identifying memory accesses of interest. Unlike traces, in case of cache side-channels, the adversary needs to control what goes into the cache. In other words, the trace-based attacks are a superset of class of attacks targeting cache side-channels. Prior work establishing metrics of information leakage such as side-channel vulnerability factor (SVF) and cache side-channel vulnerability (CSV) study the similarity between oracle and adversary traces [20, 64]. In our trace scenario, an un-obfuscated trace can be assumed as the *original trace*. Access to this by an adversary would make it easy to recover secret keys. However, there are application traces which do not embed secret keys and are still susceptible to information leakage, such as model stealing using synthetic queries [32, 36]. Other techniques such as code mutation provide security by hiding implementation details of proprietary programs [57]. Security of traces from these applications needs to be scrutinised using more generalized metrics, such as those based on entropy or distinguishability [23]. Memory trace obliviousness also explores this notion of indistinguishability, by ensuring sensitive program input does not influence the content and length of the memory access trace [39]. In such threat models, the authors observe that although the mitigation is similar to timing channels, their problems are orthogonal.

2.4 Limitation of Prior Approaches

Generating traces from synthetic inputs. To prevent passing on sensitive information, one could generate traces using synthetic inputs. However, for accurate representation of architectural behavior in the trace, such inputs must mimic the distribution of typical inputs used in real workloads, which itself is a hard problem. Consider an application that takes a social media graph, such as the one held by Facebook, as an input. Creating a synthetic version of such a graph requires

an in-depth knowledge of the input and its impact on performance, which is not only difficult for the application developers but also burdensome to update frequently, as algorithms and user behaviors change [46, 52]. Our experiments (Section 5.4) show that such an approach struggles to preserve utility.

Synthetic trace generation. Myriad techniques have been proposed under the umbrella of synthetic trace generation using statistical profiles as well as trace approximation techniques. These works are more *utility-centric*, owing to their use-case of performance optimization. One such example is the use of microarchitecture independent and dependent features to profile a trace [21, 30] and then use these profiles to generate synthetic ones which can be run on simulators or even real hardware [30]. Other profiling techniques have statistical models of structural simulation data, with the goal of improving simulation speed and accuracy [43]. There is also a large body of work which model spatial and temporal locality by generating clones [7, 10, 58–61], sometimes for specific embedded applications [29] or heterogeneous system-on-chips such as mobile devices [8]. ECS, built on WEST [10], goes further by infusing environment-specific information in these models [60]. For larger parallel machines and applications, Gao et al. [63] proposed trace approximation to compress larger memory streams. Other trace compression techniques, like PSnAP employ loop profiling for HPC applications [42]. The attributes captured in these techniques cover important program properties such as instruction mix, control flow behaviour, data locality and instruction level parallelism, which in turn influence memory footprint of an application.

Program behavior has also been characterized using other diverse approaches in prior works. Chen et al. collected samples of hardware event profiles for feedback-directed optimization [16]. Thiebaut et al. [56] modeled hyperbolic characteristics to generate synthetic memory references. SynFull focused on NoC traffic for capturing cache-coherence as well as application behavior [9]. SLAB adopted code-based correlation to identify memory access patterns [48]. PerfProx developed proxy big data benchmarks, by using hardware performance counters [46]. μsuite develops benchmarks to quantify the impact of operating system and network on microservice performance, in a data-center setting [55]. Figure 2 compares Camouflage with prior approaches based on two key attributes: (1) *Utility* validates whether an approach is verified by architectural metrics such as miss rates, speedup, (2) *Secrecy* which characterizes a framework's quantitative or qualitative justification of information present in traces. Trace-wringing [18] was one of the first approaches targeting *security*. It operates on the principles of signal processing to get structurally correct traces with a specified information transfer budget. They verify *trace security* by performing key recovery attacks on traces generated through executing AES encryptions.

Summary. Thus, traces with realistic inputs are necessary for analyzing architectural simulations; blind obfuscation diminishes their utility to actually reflect architectural behavior. On the other hand, security concerns thwart efficient trace sharing. Hence, we propose an obfuscation framework that is capable of preserving certain architectural utility, based on semantics (Section 3), without jeopardizing secrecy of input.

3 UTILITY-AWARE TRACE OBFUSCATION

Trace obfuscation, without underlying semantic knowledge, risks loss of crucial architectural behavior [18, 57]. We propose Camouflage, a program semantics-aware trace modification approach that preserves key architectural properties necessary for useful architectural simulations. The stream of memory accesses is a manifestation of the data structures used in the program. Consider, for example, two code segments: one that inserts a new element into an array and another into a linkedlist. It is assumed both the array and linkedlist are sorted in ascending order. Any insertion in the linked list would access all the nodes preceding it, followed by remapping the node

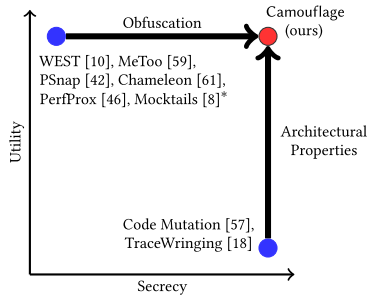


Fig. 2. Contemporary approaches in trace generation and their role in the utility-secrecy spectrum (*more details in Section 2.4).

pointers where the insertion has to occur. The rest of the list nodes occurring after the inserted node can remain as it is, hence there are no accesses streaming from that part. However, in the array-based implementation, for inserting a new entry, the elements greater than the entry have to be shifted to accommodate it. Hence, the memory access stream would reflect this semantic shift. Though this is a simple example, it illustrates how the choice and organization of data structures have a profound impact on an observed stream of accesses.

★ ***How does the use case extend to other traces?***
Although we consider properties necessary for caches, extending to other traces such as those of branch prediction is possible. This is not trivial, since the semantics governing a different architectural behavior will vary. However, the framework shown in Figure 4 and the concept of security-utility trade-off can be adopted. Note that there could be additional properties which would further improve the utility of the trace. But, these properties would have to be scrutinized for security violations as well. Given the high correlation exhibited by performance numbers between original and *camouflaged* traces (Section 5), we understand that the existing properties are sufficient to preserve *utility*.

3.1 Preserving Trace Utility

Cache behavior is reliant on inherent characteristics of an application's semantics. In this section, we define these attributes as *architectural properties*, for fundamental sequential and linked data structures. Although real programs may have more complex data organizations, these are generally derived from fundamental data structures.

Data structures with contiguous memory layout. Arrays are mapped to contiguous memory blocks. However, the memory accesses to such arrays are not always contiguous in time. Sometimes, they are dictated by input to the application, such as the S-box table in AES encryption [44].

Data-independent Array. An array has a base address for the first element and, in accordance with its predefined size and data type, a sequential chunk of memory is allocated for that array. To predict subsequent memory addresses, a prefetcher only requires the stride between two elements of the array assuming that all elements are accessed sequentially. For an array of size n with base address b and stride s, if every i^{th} element is accessed, the memory address would be the $b + i \cdot s$. Figure 3(a) depicts an array of $n = 5$ and $s = 4$ for consecutive memory accesses. Thus, we define the only architectural property as stride of the array; 4 in this example.

Data-dependent Array. Data-dependent arrays refer to those contiguous data structures where order of accesses to the array depend on the input. Alternately, they can be viewed as one-to-one hashmaps, which are frequently used in applications with large databases, owing to their storage efficiency and low search time complexity. A *key* is searched by hashing it with a predefined

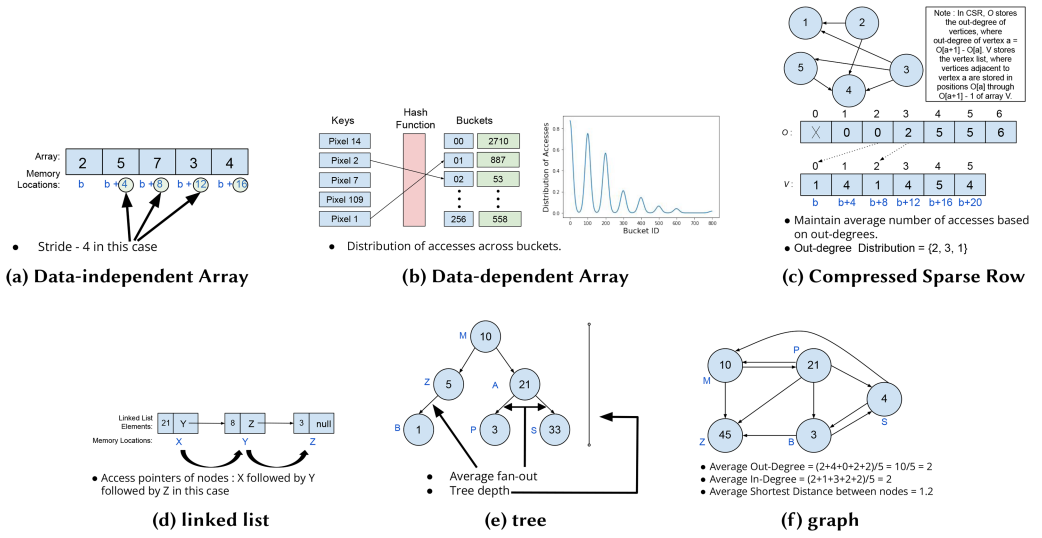


Fig. 3. Architectural properties of popular data structures.

function and using this result to locate the index of the *value* corresponding to the *key*. Given no alteration in the hash function, memory behavior is largely dependent on the input or the *key*. Figure 3(b) shows pixel values as *keys* and their corresponding frequency of occurrence in an image, as *value* in the buckets. When an image is stored in this structure, the frequency of accessing a bucket entirely depends on the pixel values. The memory sequence seen by the cache is then a function of the distribution of accesses across different buckets, which we define to be an architectural property of data-dependent arrays. If we were to mimic this distribution (shown on the right), we expect the caches to see a similar memory access pattern and exhibit similar performance. In other words, *trace utility* would be preserved if the modified trace has a similar mix of spatial and temporal distributions.

Compressed Sparse Row. CSR format of representation is extensively used in high-performance scientific computing applications to store sparse matrices, where rows are mostly populated with zeros. CSR captures column indices of non-zero elements and packs them into a dense array. This format is highly efficient for storing large matrices or graphs. It is more prefetcher-friendly, since reading vertices incur adjacent memory accesses. Figure 3(c) depicts a CSR representation of a graph. It consists of two arrays O and V: (1) O stores the out-degree of a vertex, (2) V stores the outward vertices indexed via O. In our example, when neighboring vertices are accessed, three adjacent memory accesses from the array (from b to $b+4$) are observed. To issue useful prefetches, this contiguity of accesses should be maintained in the obfuscated trace. One might argue that it can be treated equivalent to a data-independent array. However, CSR may not always incur all sequential accesses and depends on the average out-degree or average number of elements in the row of the sparse matrix. As such, we preserve the average number of temporally adjacent accesses based on this property.

Data structures with randomized memory layout. Linked data structures such as lists, trees, and graphs can be inherently allocated in non-contiguous memory locations. Thus, stride-based address prediction techniques are not always suited to such structures.

Linked list. Linked lists, unlike arrays, allow for dynamic representation in the memory, and adjacent elements need not enforce a linear memory layout. Though such a layout can be a

manifestation of the memory allocation strategy, as lists grow in size, their elements may not be spatially adjacent. The internal points are what identify the adjacent elements in this data structure. Thus, future memory accesses are dependent on what the current node is pointing to, which we define as the key architectural property for linked lists. This means that the adjacent accesses belong to elements in the same node; whereas different nodes need not necessarily be adjacent in memory. In Figure 3(d) X, Y and Z can occur anywhere in memory. However, assuming that the size of the first element is s, X, and $X + s$ has to be adjacent in the resultant trace for similar cache behavior.

Tree. Trees are often used to model applications such as file systems. We define the architectural properties of tree data structures to be (a) the average number of child nodes per node, and (b) the depth of the tree. In a tree traversal, near-future memory accesses depend on the number of child nodes and their distance from the root. For example, node 21 in Figure 3(e) points to two child nodes. The number of entries in a hypothetical prefetch table would only depend on the number of child nodes, and not the specific nodes. On the contrary, the prefetch table would look very different for a tree with an average fan-out of 10. Hence, prefetcher performance is clearly a function of average fan-out. Now, consider two trees with varying depth of 50 and 5,000. If the cache size is small, the smaller tree does not need a very efficient replacement policy, as opposed to the larger one. This observation further establishes the impact of generating traces with real-time input to obtain accurate numbers from architectural simulations. Arguably, the type of tree traversal also affects the application performance. However, dependence-based prefetchers have shown that they can adapt to different traversal orders [51].

Graph. Large graphs such as those used in web search, routing or social media analysis, have very long processing times. This can be attributed to high-latency memory accesses, which are also irregular in nature and hence harder to optimize for caches. Owing to the non-sequential nature of such workloads, stride- or stream-based caching techniques fail to improve performance of these memory intensive workloads. Even table-based prefetchers suffer from large storage cost and are unable to support beyond a certain work set size. However, prior work [6] has shown that it is possible to improve performance by considering the underlying data structure and the traversal of memory requests. In particular, we define two architectural properties that caches can leverage: (a) average in- and out-degree of the nodes, and (b) average pairwise shortest distance. Figure 3(f) shows a graph in which a search traversal node 10 will be followed by 21 and 45. From a cache's point of view, this means the stream of accesses would be $M \rightarrow P \rightarrow Z$. Thus, the number of outgoing edges determine the memory access pattern. For example, a prefetcher trained on the accesses from a graph of average degree 10, may not function the same for a graph with average degree 100. Thus, the average degree of a graph is a key architectural property affecting cache behavior.

Second, maintaining the average shortest distance between nodes ensures that memory access sequences from a depth-first traversal yield a similar context for the cache. Consider the nodes 10 and 3 in Figure 3(f), where the shortest distance between them is 2 and the memory access sequence is $M \rightarrow P \rightarrow B$. Now, a graph with a similar average shortest distance will exhibit the same memory access sequence. For an ideal prefetcher these sequences should be indistinguishable and it should be able to prefetch both of them perfectly. On the other hand, a graph with a higher average shortest distance between nodes will exhibit a longer memory access sequence. Similarly, a replacement policy will also benefit from preserving the length of an observed memory sequence.

4 CAMOUFLAGE OBFUSCATION FRAMEWORK

Camouflage operates on a *security-by-default* mechanism, such that all memory accesses are obfuscated, as if there are no known program structures in the trace. To improve utility of the trace for effective simulations, one needs to first identify data structures existent in the trace. This can

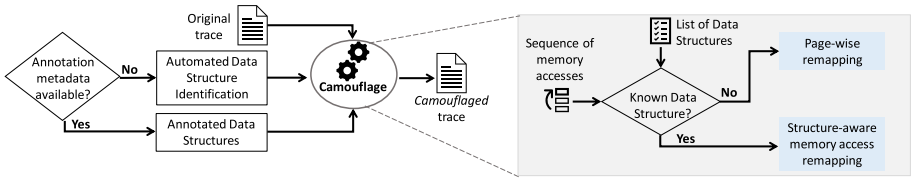


Fig. 4. Overview of Camouflage framework which takes as input an application source code and outputs an obfuscated (*camouflaged*) trace.

be done by adding annotations in the code such that they (a) immediately precede the memory allocation call of the data structure to be considered, and (b) convey the data structure type[1]. To ease the overhead associated with manually traversing the code, our framework provides an automated mechanism for data structure identification. Figure 4 shows the workflow where Camouflage relies on some form on data structure identification to obfuscate the trace. It is important to note that this metadata is optional for security concerns; in their absence, Camouflage would obfuscate the trace oblivious to program semantics.

4.1 Known Data Structure Accesses

Common Modifications. There are several side-channels that an attacker can exploit to recover information from traces, as discussed in Section 2.3. Camouflage prioritizes obfuscation of all such vulnerabilities from the original trace. Absolute address values are remapped using a random hash function[2]. Although it is imperative to remove privacy-encroaching spatial and temporal patterns, random naive obfuscation would significantly hamper the utility of the trace. Architectural artifacts such as prefetchers or replacement policies are dependent on the spatial distribution of accesses within a page. Camouflage leverages this spatial spread to redistribute accesses, such that some form of spatial adjacency is preserved at a block-level granularity In addition to this, there are some structure-specific modifications based on the properties in Section 3.1.

Data Structure-Specific Modifications

Data-independent Array. Data-independent arrays are obfuscated by remapping the array to a different contiguous memory location. Stacks and queues can also be altered using similar insights, as they have contiguous memory layout configurations. This is obviously on the higher end of the trace utility spectrum, but any other alteration destroys the inherent properties of an array Figure 5(a)).

Data-dependent Array. The Camouflage framework emulates the spatial and temporal distribution of memory access sequences for this structure. To achieve this, we first permute accesses in the spatial distribution at a 64B cache block granularity. We then generate a temporal distributions of accesses. Following which, we use statistical modelling and draw a finite set of samples from both the distributions.

Compressed Sparse Row. Apart from re-mapping like a data-independent array, we obfuscate the exact number of accesses in an interval to prevent giving out information about the input. To do this, we obtain the distribution of accesses based on out-degrees. Then, we re-sample from this distribution such that the average out-degree stays the same. Figure 5(c) shows that the average

[1]For some structures, memory access sequences are also impacted by the traversal order in the data structure, such as BFS or DFS in graphs. We provide more details of this in Section 4.1.
[2]The function is kept secret. This can be done by using a popular hash function, such as SHA-256 and using a secret salt which is deleted after the trace generation. Or using linear-feedback shift register (LFSR) with a secret initial state.

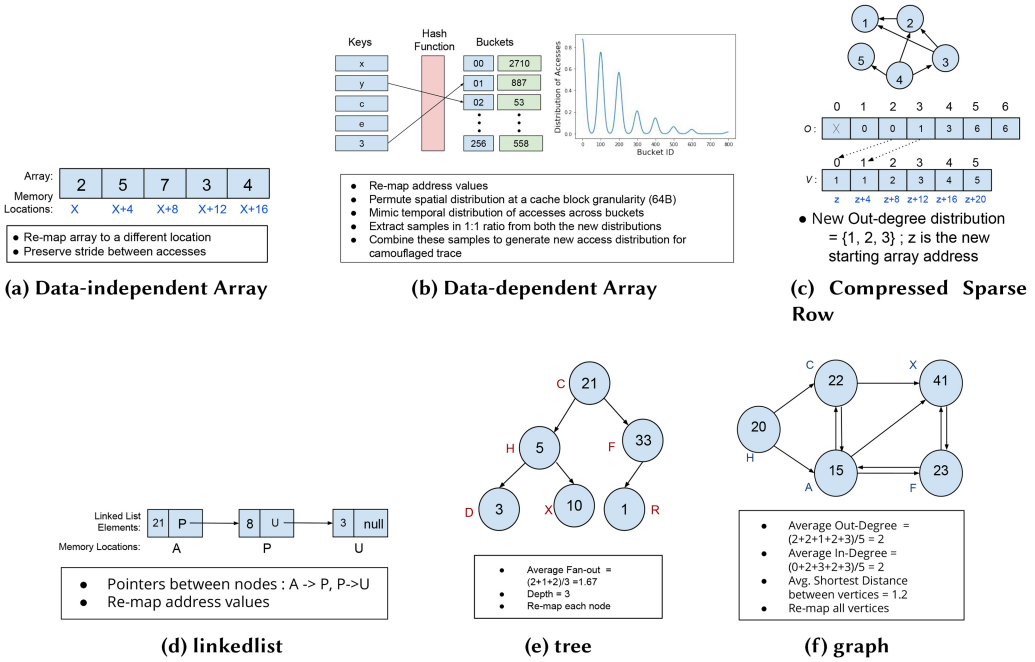


Fig. 5. Data structures as seen in a camouflaged trace. Certain architectural properties such as stride in array and in-/out-degree distribution in graphs are preserved by Camouflage.

out-degree is 2, which is the same as in Figure 3(f), however the distribution has changed, which in turn reflects the ordering of memory accesses for array O and V. A similar notion applies to sparse matrices represented in this format as well.

Linked Structures (such as Linkedlist, Tree, Graph). Due to lack of contiguity, linked structures can be obfuscated as long as links between elements of sequences are maintained in a manner similar to the original trace. For **Tree**, memory sequences indicating average fan-out and depth, should be preserved in the *camouflaged* trace. **Graphs** have an additional attribute: traversal type. The camouflaging process preserves this for retaining program behavior. The synthetic sequence of memory accesses for a graph can be visualized as a restructured graph, of accesses, such that average degree is preserved. For example, in Figure 3(f) the sequence M follows $P \rightarrow Z$ which follow $B \rightarrow S$ in a BFS traversal. However, in the average case scenario, *camouflaged* trace can have two accesses followed by three. This is showcased in Figure 5(f) where H is followed by $C \rightarrow A$, followed by $X \rightarrow F$, thus the overall graph does look like a different one than Figure 3(f). Note that, this example is only for demonstration and must not be mistaken as an exchange of access sequence lengths. The averaging process is applied to very large social media graphs and encompasses the entire trace.

4.2 Auxiliary Accesses

Each data structure has a unique set of architectural properties. Camouflage processes each of them every time it hits a programmer hint. However, there are other memory accesses, which can leak subsidiary information. This de-prioritization in trace security must be prevented. For maximum trace security, this obfuscation would mean re-mapping every access such that all unnecessary spatial and temporal patterns are obviated. However, this diminishes trace utility as

prefetcher performance depends on number of pages seen by it. To trade off utility and secrecy, Camouflage remaps memory addresses such that (1) number of pages stay constant across original and *camouflaged* traces, (2) accesses are uniformly distributed across all pages, ensuring all *camouflaged* traces for different inputs look indistinguishable.

★ ***Why does page remapping not mislead hardware design?***
Unannotated accesses constitute < 30% of the trace, which being a much smaller fraction, does not impact the average performance, as it is not identified as a bottleneck. Additionally, most cache and memory techniques often operate at higher levels in the cache hierarchy and thus deal with physical addresses where contiguity is limited. As such applying a page-aware absolute address randomization does not mislead cache utility.

★ ***What about missed data structures?***
Identifying all structures which contribute to performance is a key feature in preserving utility in Camouflage. Hence, missed annotations would impact the performance measurements. However, they do not lower trace security as Camouflage is *security-by-default* and obfuscates all accesses, in the absence of annotations. This motivates us to design an easier average-case obfuscation, in the next section.

4.3 Camouflage-auto Framework

Manual annotation can be tedious, especially for large code bases. Although, programmers need to do this annotation only once, before generating the *camouflaged* traces, we realize automating the process will make Camouflage more adaptable. We are motivated by prior approaches for reverse engineering data structures from binaries [26, 54]. In the context of memory access traces, several machine learning techniques have been explored, especially for studies around prefetching and replacement [40, 50, 53]. Itai and Slavkin applied machine learning to identify data structures in an address trace, however, their approach assumes singular data structure and very simple operations on it, as opposed to traces from benchmarking applications [27]. We design Camouflage-auto as a combination of partitioning and classification techniques to determine data structures in the trace. It then proceeds with trace obfuscation according to Camouflage framework, based on the detected data structures.

Automatic Data Structure Identification

Partitioning. To identify data structures, we first segment the trace into different process sections, such as code, heap, and stack accesses. Next, we group memory addresses based on 32 MSBs of the addresses, which we call memory partitions. We found based on our workloads that a single partition typically has one type of data structure. Although this might not be true for all applications, we do observe our set of workloads complying to this heuristic. For each partition we compute the distribution of the differences in consecutive memory accesses. We then create a feature vector $\mathbf{x}$ of size 17 for a partition such that $\mathbf{x}_i$ denote the fraction of consecutive memory accesses inside a partition that differ by less than 2^i but no less than 2^{i-1}, where $0 \leq i \leq 16$.

Classification. We train a multi-class **multi-layer perceptron (MLP)** classifier to predict the data structure, given a feature vector of a partition. The MLP has two hidden layers with ReLU activation and an output layer with softmax activation. We created a dataset of feature vectors of the partitions, with our hand-labeled data structures. We segregate our benchmarks into train and test sets, such that both encompass all known data structures, as shown in Table 1. We achieve an average of 95% and 82% accuracy on the training and test set, respectively. We observe that the mis-classified test cases primarily belong to graph or CSR, predicted as a data-independent array. Given the predominance of samples from arrays in the training set, this observation is justified.

Clustering. Although classification assigns labels to a partition, misclassification might yield adjacent partitions to have different labels. If a spatial region is dominated by a certain label, it is more likely, the intermediate partitions belong to the label with the majority and were misclassified. Hence, we apply an agglomerative clustering approach, with distance-based metric, to identify unique clusters among feature vectors, to assess a majority label. Following which, we apply a filtering technique to weed out inconsistent predictions for partitions belonging to the same cluster.

Although, Camouflage-auto does a good job for our set of applications and its ease of use makes it a viable option for trace obfuscation. However, for the most accurate performance characterization, we recommend usage of Camouflage, especially in cases where the programmer is aware of the bottleneck areas of the code.

5 EVALUATING UTILITY OF CAMOUFLAGED TRACES

This section details our choice of metrics that reflect trace *utility* as well as the experimental setup used for adjudging these metrics. We evaluate how well trace generators can preserve architectural behaviors seen in original traces. For comparison to prior work, we use traces modified using the Mocktails [8] and Trace-wringing [18].

5.1 Measuring Trace Utility

We measure the utility of a trace by comparing the speedup across different prefetchers and replacement policies. For prefetcher variation, we also report percentage of useful prefetches. For replacement policy variation, we look at last level cache miss rates. The numbers seen by the original trace will henceforth be referred as "original utility".

Prefetcher Variation

Speedup. Prefetchers analyze memory access patterns to reduce memory latency. As such, adjudging accurate memory behavior representation in a synthetic trace can be done by comparing prefetcher speedup of the original trace. Prefetcher speedup is calculated by normalizing **IPC (Instructions per Cycle)** to a baseline system with no prefetcher enabled. To maintain *trace utility* we look at this speedup trend before and after obfuscation. For example, if a prefetcher (f_A) performs better than another prefetcher (f_B) on the original trace, a good trace modification technique would preserve the similar trend. Given the degree of obfuscation introduced in the synthetic trace, absolute prefetcher speedup numbers may not always be preserved. We adopt a ranking metric based on correlation between two lists to represent this speedup comparison (Section 5.3). For our evaluation, we use the following prefetchers: IP-Stride [45], **Kill the program counter (KPCP)** [35], Next-line [31], Pangloss [49] and **Signature path prefetcher (SPP)** [34]. The prefetchers are selected based on their occurrence in actual systems and good performance numbers exhibited in prefetching championships [5]. For this study, the replacement policy adopted across all cache levels was LRU.

Percentage of useful prefetches. In addition to speedup, we look at the variation in this metric, which is calculated if a prefetched data is actually consumed. Since our prefetchers are enabled at L2, we report the ratio of useful prefetches at this cache level. We report the difference in percentage of useful prefetches of an obfuscation technique w.r.t. the original.

Replacement Policy Variation

Speedup. Replacement policies aim to better manage the **last-level caches** [2] **(LLCs)**, such that data is not evicted too early or too late. For our study, we consider **Random Replacement (RR)** as the baseline replacement policy. With no prefetchers enabled, we evaluate the speedup over this baseline for four other replacement policies — LRU, SHIP [62], SRRIP, and DRRIP [28]. If one

Table 1. Known Data Structures in Each Application with Train-Test Split for Camouflage-auto

	Applications	Data structures
Train	deepsjeng	Data-dependent Array, Linked list
	gcc	Graph, Data-independent Array
	lbm	Data-independent Array
	x264	Data-dependent Array
	xalancbmk	Graph
	bc, bfs, cc, tc	Compressed Sparse Row
Test	mcf	Data-independent Array, Graph
	histeq	Data-independent & Data-dependent Array
	xz	Data-independent Array
	pr, sssp	Compressed Sparse Row

replacement policy (say LRU) performs better than another (say SHIP) for the original trace, we study how closely our obfuscated traces follow this "original utility". To quantify this, we use a correlation coefficient (Section 5.3).

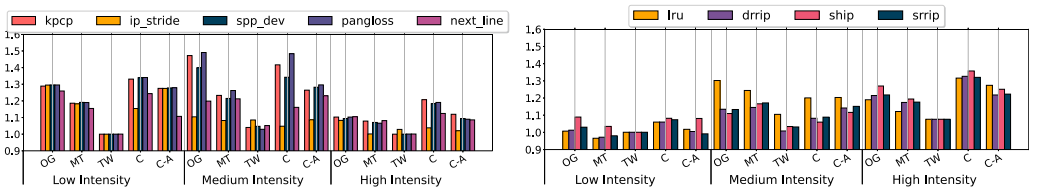
LLC Miss Rate. We look at LLC miss rates, normalized to the baseline policy. To better understand this, we project a heatmap with each cell representing the difference pre and post obfuscation. A lower difference indicates a closer number to the "original utility".

5.2 Experimental Setup

We capture program behavior for seven memory intensive applications from the SPEC2017 benchmark suite [14], histogram equalization from the PERFECT suite [11] and six graph algorithms from the GAPBS suite [12]. The traces used in our evaluation are profiled using a dynamic instrumentation tool, PIN [41]. Each application is instrumented to capture the trace structure discussed in Section 2.1, to be compatible with ChampSim [3]. It is a microarchitectural trace-based simulator used in 3rd Data Prefetching Championship [5] and 2nd Cache Replacement Championship [2]. We use a 512KB 8-way associative cache for our simulations. Traces tend to grow very large, which is why we used pre-defined simulation points, used by traces in [5]. We trace about 100 million instructions at these simulation points and then use Camouflage to modify the trace, based on programmer annotations. For example, if the programmer flags an array, then modification techniques as per Section 4 are applied. The rest of the trace is remapped in a page-aware manner Section 4.2. For each application, we identified the most memory intensive data structures and manually annotated them. The key data structures for our workloads are shown in Table 1. Camouflage is equipped to handle multiple data structure types and well as multiple structures of same type, in any application. For processing synthetic addresses, we use a pseudo-random number generator implemented by a **linear feedback shift register (LFSR)**.

5.3 Performance Analysis

We devise a ranking system for prefetcher performance using the Kendall rank correlation coefficient (τ) [33] between the original and *camouflaged* traces. τ captures pairwise disagreement between two ranked lists. We compute the speedup for traces, before and after parsing through an obfuscation, across prefetchers and replacement policies stated before. Then we rank the different policies based on the their speedups for both the traces. Finally, we compute the τ between these two ranked speedup lists. A τ closer to 1 indicates a greater similarity between two ranked



(a) Prefetcher (b) Replacement

Fig. 6. Performance comparison of prefetching and replacement policies for different traces: Original (OG), Mocktails (MT), Trace Wringing (TW), Camouflage (C) and Camouflage-Auto (C-A). The speedup is normalized to a baseline with prefetching disabled and LRU replacement, for prefetcher trends and, with random replacement baseline and no prefetching for cache replacement.

lists, which translates to lower variation in speedup trends across pre and post-camouflaged traces. Hence, for better trace utility τ should be closer to 1.

For reporting speedup numbers, we categorize the applications in three broad groups, based on their degree of memory intensive nature, according to **misses per kilo instruction (MPKI)**: (a) low intensity (MPKI $\leq$ 1): histeq, bc; (b) medium intensity (1 $<$ MPKI $<$ 10): gcc, lbm, x264, xalancbmk; and (c) high intensity (MPKI $\geq$ 10): deepsjeng, mcf, xz, bfs, cc, pr, sssp, tc. Figure 6 depicts the average speedup in each intensity group. In both prefetcher and replacement studies, the workloads follow the speedup trends very closely for Camouflage, Camouflage-auto and Mocktails. The effect is less pronounced in a high intensity workload group. Figure 8(a) plots the absolute difference in percentage of useful prefetches of obfuscated traces using Camouflage, Camouflage-auto, Mocktails and Trace Wringing, w.r.t. original traces. A darker area indicates a larger difference in this metric, i.e., worse utility. Both traces from Camouflage and Camouflage-auto do not exhibit any notable difference, apart from deepsjeng. In deepsjeng, we find that different prefetchers yield similar speedups and thus the number of useful prefetches sees little variation across different trace generators. A similar utility study is performed for varying replacement policies, shown in Figure 8(b), where *camouflaged* traces maintain very similar LLC miss rates compared to the other approaches, indicating good utility. *xz* exhibits a slightly higher delta of miss rate for Camouflage-Auto. Since this is a data compression application, it is possible that the array data structure was mis-classified as multiple arrays instead of one large array, due to the automated access pattern detection nature of Camouflage-auto. When the *camouflaged* trace was generated with this notion, the arrays were allocated in different memory regions, thus altering the miss rate. Camouflage-Auto is an automated process and to get more accurate obfuscation, usage of Camouflage is recommended.

Comparison with Prior Works. We compare both versions of Camouflage with Mocktails [8] and Trace Wringing [18]. For Mocktails, we adopt the two level hierarchy with temporal followed by dynamic spatial partitioning. Mocktails dynamically identifies spatial regions to generate memory models which is then used to replay the original workload behavior. For Trace Wringing, we try to tune the progressive probabilistic Hough transform parameters to the best of our ability. On observing the τ coefficient numbers in Figure 7, Mocktails exhibits a relatively lower correlation with the original trace, as opposed to Camouflage. Figure 8(a) also shows Mocktails-generated traces from histeq and xz exhibit larger variation for useful prefetches. Trace Wringing exhibits negative correlation in speedup for some workloads and relatively lower correlation numbers when compared to Mocktails, Camouflage and Camouflage-auto (Figure 7). This happens because most prefetchers don't exhibit significant relative speedup over the baseline, as seen in Figure 6. The τ numbers seen in Figure 6(b) align with the trends observed in LLC miss rate in Figure 8(b).

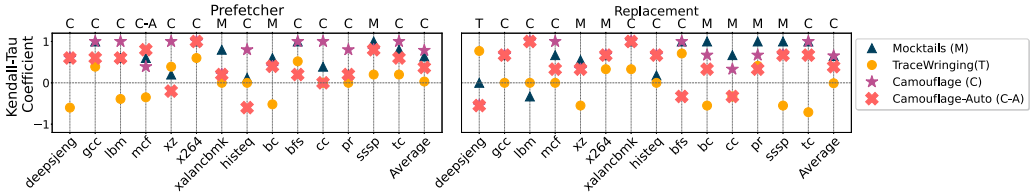
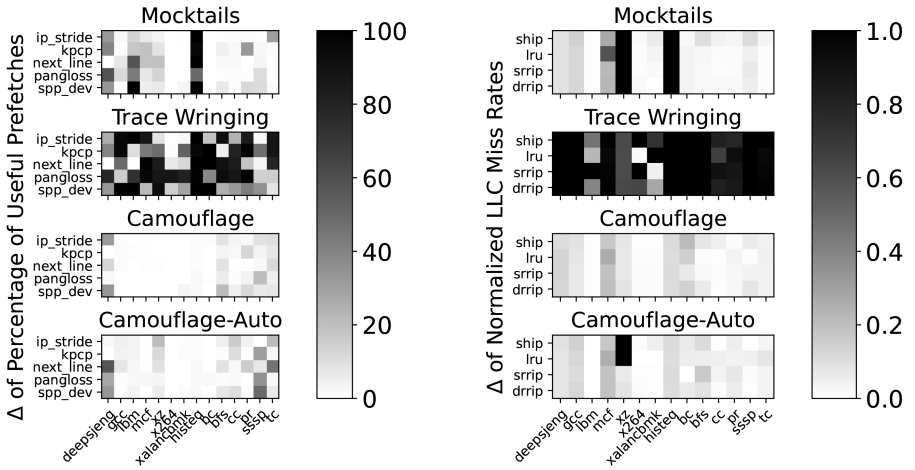


Fig. 7. The Kendall Tau coefficient (τ) [33] of IPC speedup of synthetic traces generated by Camouflage, Camouflage-Auto, Mocktails and Trace Wringing with respect to the "original" trace for different applications. $\tau \in [-1, 1]$, where $\tau = -1$ means least correlation. The top row indicates the obfuscation technique with the highest correlation.



(a) Useful Prefetches (b) LLC Miss Rate

Fig. 8. Heatmaps showing two different utility metrics, under different trace obfuscation schemes. A darker cell means a higher difference and thus worse utility of the obfuscated trace.

5.4 Additional Utility Comparison using Synthetic Inputs

Creating synthetic input for real programs seem to be a contrastive but plausible approach for secrecy problems, provided the inputs are easily generated and do not demand in-depth knowledge of the application. For this study, we use a graph generator provided with GAPBS [12] to create uniform random graphs with a number of vertices equal to that of actual graphs of social networks, roadmaps and web graphs from SNAP [38]. Following which we generate traces for 10 synthetic graphs and compare the speedup relative to the original trace, in terms of τ. We observe $-0.19 < \tau < 0.39$ across all GAPBS applications. This is significantly lower than Camouflage, which ranges between 0.39-0.99 across GAPBS applications, as shown in Figure 7.

6 EVALUATING SECURITY OF CAMOUFLAGED TRACES

Camouflage carefully obfuscates traces to remove information about the private input in the trace, while preserving their architectural properties. By default, Camouflage obfuscates all memory accesses using a page-aware remapping of all memory addresses, unless the access is part of a data structure that is annotated by the developer — in that case it uses tailored obfuscation techniques that preserve the architectural properties in the trace (as shown in Section 3). Although such

tailored obfuscation is good for utility of the trace, it can still compromise the private input. In this section, we measure how well Camouflage protects the inputs used for generating a trace.

Measuring secrecy of the inputs is not trivial, because even few bits of information about the input can be damaging. Therefore, instead of information leakage or entropy-loss based metrics used in prior works, we follow the definition used in the cryptography community to analyze the security of message encryption schemes, namely message indistinguishability [24] and message recovery [13] security. We analyze input secrecy in two different ways: (a) input indistinguishability, and (b) resilience to input recovery.

6.1 Measure Input Secrecy via Input Indistinguishability

The input indistinguishability of a trace generation technique is defined as follows: Given a trace generated from a randomly sampled input $\gamma \in \mathcal{I}$ from a finite set of inputs $\mathcal{I}$, an attacker is tasked with identifying the input that was used to generate the trace. For a secure trace generation procedure the attacker should not be able to succeed in this task (with probability better than random guessing). The set of inputs $\mathcal{I}$ is chosen to match the realistic distribution of inputs (to avoid degenerate inputs, such as all zeros, that produce obviously distinct traces). In our threat model, the attacker knows the set of inputs $\mathcal{I}$ and has access to the trace generation and obfuscation frameworks, thus the attacker can generate as many traces of those inputs as computationally feasible. The only information the attacker does not know is the randomness used during the generation process of the challenge trace. Input (in)distinguishability property captures *any* leakage of sensitive information via the trace. For example, an attacker who can extract any non-trivial information about the input from a given trace must be able to at least identify which input was used to generate the trace, and therefore will be able to break the input indistinguishability property.

Following this definition, the attacker's strategy can be modelled as a multi-class classification task, where the attacker trains a classifier to classify traces into one of the input classes $\mathcal{I}$. The adversary thus takes a trace as input and outputs the identifier of the input used to generate the trace. We quantify this indistinguishability in terms of *security loss*, measured as the difference between the achieved classification accuracy and the accuracy of random guessing (which is $1/|\mathcal{I}|$). Hence, a system is deemed ideally secure if its security loss is 0, where the adversary's classification accuracy is close to the probability of a random guess.

Selecting Features and Training Classifiers to Distinguish Inputs.

Given a trace, we therefore first extract a set of features based on the spatial distribution of access patterns in the trace; such access pattern has shown to reveal information about the input. Specifically, we start by normalizing the memory addresses in a trace by dropping the least significant 6 bits — assuming 64-byte cache blocks — and computing the relative distance with respect to the lowest memory block address present in the trace. Most systems operate on a cache block size of 32 – 128 bytes. During the process of judging indistinguishability and feature vector extraction, an adversary can experiment with multiple block sizes to select the best one. In our study, 64 bytes is the empirical optimum. We only consider the smallest n normalized memory locations that are accessed at least once in the trace. Then, for each of the n normalized block addresses, we compute the fraction of memory accesses that involve that block. This spatial distribution of accesses to n memory locations is used as the feature vector for our classifier. We do not know the optimality of these features; there might exist a different set of features that can distinguish *camouflaged* traces better using trace distinguishability. However, to the best of our knowledge this is the first exploration of how we can analyze security of obfuscated traces, and this combination of features provided a good overall performance on distinguishing traces.

It is unclear what strategy an attacker could take to maximize their accuracy this classification task, especially given the large size and complexity of the trace datasets. Therefore, to learn the

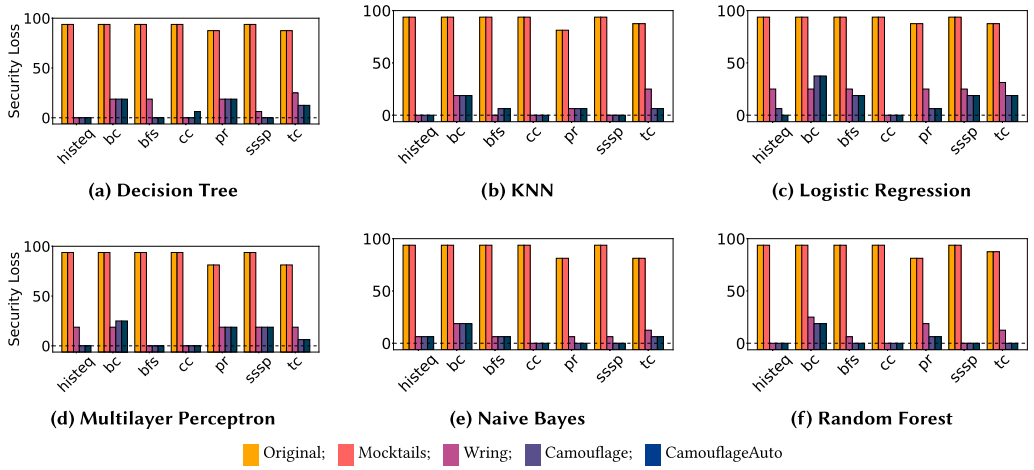


Fig. 9. Security loss observed across different classifiers to identify inputs from a given trace of different types of workloads. The dotted line shows minimum security loss, where the classifiers' success is equal to that of a random guess.

accuracy of multi-class classification problem of identifying the input (class) given a trace, we considered six popularly used classification algorithms: **decision tree (DT)**, *k*-**nearest neighbor (KNN)**, **logistic regression (LR)**, **multi-level perceptron (MLP)**, **Naive Bayes (NB)** and **random forest (RF)**. Although this approach does not provide an absolute theoretical upper bound on the attack's success, it provides an empirical understanding of the accuracy that an attacker could achieve in practice. Each classifier takes the feature vector explained above, and outputs a value between $\{1, 2, \ldots, |\mathcal{I}|\}$, denoting the class label of the input. As the feature dimension is quite large, we reduced the dimension to 100, using **principal component analysis (PCA)**, prior to feeding the training data into the classifier. For the security analysis, we consider six applications from GAPBS workloads and histeq. We do not consider SPEC17 workloads as they take specific scientific inputs which are hard to obtain. On the contrary, GAPBS takes graphs as inputs, which we obtained from SNAP datasets [38]. We also consider histeq. Input images for histeq were obtained by processing 16-bit pixel images of 3840×2160 resolution from the web.

For each workload, we choose $|\mathcal{I}| = 16$ different inputs and generate 10 traces for each input for each workload. For each input we use eight randomly chosen traces for training, and keep two traces for testing. We train the classifiers using these $8 \times 16 = 128$ traces, and test with the remaining 32 traces.

In our case, minimum security is 6.25%, assuming equal probability across 16 labels. We compare the security loss of Camouflage and Camouflage-auto with that of Mocktails [8] and Trace Wringing [18] and report the numbers in Figure 9. For the baseline comparison, we also compute the classifier's performance on the original trace (without any obfuscation).

★ ***Why do we restrict our evaluation to 16 classes and 10 traces per input?***
In real-world situations the input can be from a large space (much bigger than 16), adding more complexity to the attacker's task. Thus, our results provide a conservative upper estimate of the attacker's success. We chose 16 to balance between computational overhead while also gaining reasonable insight into the attacker's success probability. We limit our analysis to 10 traces per input, because generating traces from SPEC benchmarks and large graph workloads is computationally expensive. Additionally, in order to train classifiers on them, these traces need to be stored

and processed. In our research machine (with 16 core, 12^{th}-gen Intel Core i9 processor), generating 160 traces for a workload takes on average 50 hours.

Results. As shown in Figure 9, original traces (without any obfuscation) are easy to distinguish; all classifiers achieve near perfect security loss (93.8%) for most of the benchmarks. Traces generated using Mocktails are also easy to identify. Our classifiers achieve an average of 97.2% accuracy (which amounts to 91% security loss) across all benchmarks. This is expected, as Mocktails [8] focus on preserving *trace utility* as opposed to the security. Traces modified by Trace Wringing [18] exhibit significantly lower security loss, with average classification accuracy of 17%. Again, this is due to their custom construction that limit the information passed onto the resultant trace. (However, this also hampers their utility as we show in Section 5.)

For almost all classifiers and across all benchmarks, Camouflage and Camouflage-auto traces exhibit similar performance, and much lower average security loss than all, 6%. Overall, Logistic regression perform the best. For some workloads such as bc, both Camouflage and Camouflage-auto performs poorly; the classifier can identify the correct input with 37% accuracy. We are not exactly sure about the reason for this; future works can investigate the effect of workloads on trace obfuscation and input secrecy.

6.2 Resilience to Input Recovery

In Section 6.1, we show that Camouflage traces resist identifying the input from a small set of inputs. Here, we investigate the input recovery security through a set of known attacks against traces. We show that traces generated using prior work can be analyzed to extract keys, while it is not possible to do so for *camouflaged* traces. We did not investigate new attacks that could be possible on *camouflaged* traces nor do we claim that Camouflage traces are secure against more sophisticated attacks. Indeed future work should investigate developing new attacks against trace obfuscation techniques like Camouflage. We will therefore open source all code and data we used in this paper to encourage follow up work on trace security. Here we demonstrate one piece of evidence that Camouflage is able to prevent known input recovery attacks against memory access traces.

AES key recovery.

It is possible to retrieve keys from a trace of AES encryption if the adversary knows the corresponding plaintext and the ciphertext. We adopt this simple attack from [18, 44]. The AES encryption includes a non-linear transformation step performed by accessing the Rijndael substitution function (S-box), which is stored in the memory as a table. In the first round, offsets used to index the table are determined by XOR-ing each key byte with the plaintext. Hence, discovery of offsets reveals the key used for encryption, given the adversary knows the plaintext. The memory access sequence can be easily pruned to extract addresses of interest to the attacker. Following which, the attacker tries to determine the base address of the S-box table and get the offset. This offset when XOR-ed with the plaintext reveals the key. However, if these offsets were obfuscated, it would be difficult to recover correct key bytes from the first round of encryption. Additionally, it would be nearly impossible to get the key, unless the first round succeeds, as subsequent rounds depend on the state of the first round. Camouflage takes advantage of such semantic insights to obfuscate accesses.

We verify if Camouflage can prevent such attacks on encryption applications. We perform the first-round-attack on each of these applications. We assume that they store the tables in memory, similar to the strategy adopted in [18, 44]. We applied the attack on the traces generated by Camouflage for AES, DES, and Blowfish. We also measure trace utility, for all the three encryption applications, using the metrics in Section 5.1. There is no significant change in speedup numbers, because none of these applications are memory intensive. The tables in memory are 512B in size,

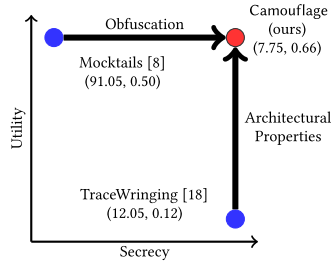


Fig. 10. Summary of secrecy-utility results, shown as tuples of average security loss and average τ.

which fits easily in the L1 cache. We found the attack cannot retrieve the key for all three applications. For comparison, we tested the same attack on the traces generated by Mocktails and Trace Wringing and the key was not recoverable.

7 SECRECY-UTILITY TRADEOFF IN CAMOUFLAGE

Camouflage provides a tunable balance between secrecy of user inputs and the utility of the trace. We show the *utility* (measured as average τ) and *secrecy* (measured as average attack failure rate) of different trace generation techniques in Figure 10. Mocktails performs better for utility but tends towards maximum security loss. Whereas Trace Wringing provides better secrecy, but fails to preserve utility. Camouflage outperforms these trace generation techniques when we consider both the utility and secrecy of the trace. Even Camouflage-auto fares very well on secrecy, with an average security loss of 7.75%, as it leverages the conservative crux of our obfuscation strategy. It suffers slightly on the performance side as seen in Figure 7. However, we do believe that is acceptable. Thus, Camouflage provides a novel way of generating useful traces for performance optimization whilst assuring minimal information leakage.

8 CONCLUSION

Trace utility and secrecy are rarely explored in the same context as they operate on either end of the spectrum. Camouflage navigates through the challenges involved in synthetic trace generation for traces of workloads which are never released, owing to risks associated with user privacy. Although we focus on specific architectural behaviors, the framework could be extended to other traces, where the utility-governing properties would depend on the behavior in the study. Since we design Camouflage to be *security-by-default*, no structure would be able to unknowingly reveal information. Our experiments on trace security show low success rates of an attack by an adversary, thus proving mitigation of latent information. We believe such an obfuscation framework can facilitate a more inclusive set of applications, for representative and timely hardware design space exploration.

REFERENCES

[1] 2016. Championship branch prediction (CBP-5). (2016). https://jilp.org/cbp2016/
[2] 2017. Cache replacement championship (CRC-2). (2017). https://crc2.ece.tamu.edu/
[3] 2017. ChampSim. (2017). https://github.com/ChampSim/
[4] 2018. Championship value prediction (CVP-1). (2018). https://www.microarch.org/cvp1/cvp1online/program.html
[5] 2019. 3rd Data Prefetching Championship. (2019). https://dpc3.compas.cs.stonybrook.edu/?final_programs
[6] Sam Ainsworth and Timothy M. Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. Article 39.

[7] Amro Awad and Yan Solihin. 2014. STM: Cloning the spatial and temporal memory access behavior. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 237–247.
[8] M. Badr, C. Delconte, I. Edo, R. Jagtap, M. Andreozzi, and N. E. Jerger. 2020. Mocktails: Capturing the memory behaviour of proprietary mobile architectures. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 460–472.
[9] M. Badr and N. E. Jerger. 2014. SynFull: Synthetic traffic models capturing cache coherent behaviour. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 109–120.
[10] G. Balakrishnan and Y. Solihin. 2012. WEST: Cloning data cache behavior using Stochastic Traces. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12.
[11] Kevin Barker, Thomas H.Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. 2013. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute. http://hpc.pnnl.gov/projects/PERFECT/
[12] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. https://doi.org/10.48550/ARXIV.1508.03619
[13] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. Message-locked encryption and secure deduplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 296–312.
[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. 41–42.
[15] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. 2021. SoK: Design tools for side-channel-aware implementions. *arXiv preprint arXiv:2104.08593* (2021).
[16] 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* (Toronto, Ontario, Canada) *(CGO '10)*. 42–52.
[17] Mia Xu Chen, Benjamin N. Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M. Dai, Zhifeng Chen, Timothy Sohn, and Yonghui Wu. 2019. Gmail smart compose: Real-time assisted writing. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 2287–2295. https://doi.org/10.1145/3292500.3330723
[18] Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. 2019. Safer program behavior sharing through trace wringing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1059–1072. https://doi.org/10.1145/3297858.3304074
[19] Deeksha Dangwal, Zhizhou Zhang, Jedidiah R. Crandall, and Timothy Sherwood. 2021. Context-aware privacy-optimizing address tracing. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. 150–162. https://doi.org/10.1109/SEED51797.2021.00027
[20] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 106–117. https://doi.org/10.1109/ISCA.2012.6237010
[21] L. Eeckhout, K. de Bosschere, and H. Neefs. 2000. Performance analysis through synthetic trace generation. In *2000 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS (Cat. No.00EX422)*. 1–6.
[22] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. 2010. Improved trace-driven cache-collision attacks against embedded AES implementations. In *International Workshop on Information Security Applications*. Springer, 243–257.
[23] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (may 1996), 431–473. https://doi.org/10.1145/233551.233553
[24] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *Journal of Computer and System Sciences* 28, 2 (1984), 270–299.
[25] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 655–672.
[26] I. Haller, A. Slowinska, and H. Bos. 2013. MemPick: High-level data structure detection in C/C++ binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 32–41. https://doi.org/10.1109/WCRE.2013.6671278
[27] Alon Itai and Michael Slavkin. 2007. Detecting data structures from traces. In *Proceedings of the Workshop on Approaches and Applications of Inductive Programming, AAIP'07, September 17, 2007, Warsaw, Poland*, Emanuel Kitzelmann and Ute Schmid (Eds.). 39–50. https://cogsys.uni-bamberg.de/events/aaip07/aaip_print.pdf#page=47
[28] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer*

Architecture (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[29] A. Joshi, L. Eeckhout, R. H. Bell, and L. John. 2006. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *2006 IEEE International Symposium on Workload Characterization*. 105–115.

[30] A. Joshi, L. Eeckhout, and L. John. 2008. The return of synthetic benchmarks. Presented at the SPEC Benchmark Workshop, San Francisco, CA, USA, (2008), pp. 1–11.

[31] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. 364–373.

[32] M. Juuti, S. Szyller, S. Marchal, and N. Asokan. 2019. PRADA: Protecting against DNN model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 512–527. https://doi.org/10.1109/EuroSP.2019.00044

[33] M. G. Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1-2 (06 1938), 81–93. https://doi.org/10.1093/biomet/30.1-2.81 arXiv:https://academic.oup.com/biomet/article-pdf/30/1-2/81/423380/30-1-2-81.pdf

[34] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[35] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. 737–749.

[36] Taesung Lee, Benjamin Edwards, Ian Molloy, and Dong Su. 2018. Defending against model stealing attacks using deceptive perturbations. *CoRR* abs/1806.00054 (2018). arXiv:1806.00054 http://arxiv.org/abs/1806.00054

[37] Victor Lee, Derek Bruening, and Parthasarathy Ranganathan. 2022. *Google Workload Traces 2022*. https://research.google/resources/datasets/google-workload-traces-2022/

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data

[39] Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2013.11

[40] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 579, 11 pages.

[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[42] Catherine Mills Olschanowsky, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PSnAP: Accurate synthetic address streams through memory profiles. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin, Berlin, 353–367.

[43] M. Oskin, F. T. Chong, and M. Farrens. 2000. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 71–82.

[44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Cryptographers' track at the RSA Conference*. Springer, 1–20.

[45] S. Pakalapati and B. Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 118–131.

[46] Reena Panda and Lizy Kurian John. 2017. Proxy benchmarks for emerging big-data workloads. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 105–116. https://doi.org/10.1109/PACT.2017.44

[47] Reena Panda and Lizy K. John. 2018. HALO: A hierarchical memory access locality modeling technique for memory system explorations. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 118–128. https://doi.org/10.1145/3205289.3205323

[48] R. Panda, X. Zheng, and L. K. John. 2017. Accurate address streams for LLC and beyond (SLAB): A methodology to enable system exploration. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 87–96.

[49] Philippos Papaphilippou, Paul H. J. Kelly, and Wayne Luk. 2019. Pangloss: A novel Markov chain prefetcher. *CoRR* abs/1906.00877 (2019). arXiv:1906.00877 http://arxiv.org/abs/1906.00877

[50] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. 285–297.

[51] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA). 115–126.

[52] Anupam Sanghi, Raghav Sood, Dharmendra Singh, Jayant R. Haritsa, and Srikanta Tirthapura. 2018. HYDRA: A dynamic big data regenerator. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1974–1977. https://doi.org/10.14778/3229863.3236238

[53] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3352460.3358319

[54] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society. https://www.ndss-symposium.org/ndss2011/howard-a-dynamic-excavator-for-reverse-engineering-data-structures

[55] A. Sriraman and T. F. Wenisch. 2018. μSuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12.

[56] D. Thiebaut, J. L. Wolf, and H. S. Stone. 1992. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.* 41, 4 (1992), 388–410.

[57] Luk Van Ertvelde and Lieven Eeckhout. 2008. Dispersing proprietary applications as benchmarks through code mutation. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. 201–210.

[58] Y. Wang, A. Awad, and Y. Solihin. 2017. Clone morphing: Creating new workload behavior from existing applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 97–108.

[59] Y. Wang, G. Balakrishnan, and Y. Solihin. 2015. MeToo: Stochastic modeling of memory traffic timing behavior. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 457–467.

[60] Y. Wang and Y. Solihin. 2015. Emulating cache organizations on real hardware using performance cloning. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 298–307.

[61] 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* (Island of Kos, Greece) *(ICS '08)*. Association for Computing Machinery, New York, NY, USA, 36–45. https://doi.org/10.1145/1375527.1375536

[62] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 430–441.

[63] Xiaofeng Gao, A. Snavely, and L. Carter. 2006. Path grammar guided trace compression and trace approximation. In *2006 15th IEEE International Conference on High Performance Distributed Computing*. 57–68.

[64] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B. Lee. 2013. Side channel vulnerability metrics: The promise and the pitfalls. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2487726.2487728

Received 20 October 2023; revised 1 February 2024; accepted 6 February 2024