Side-channel Vulnerability Factor: A Metric for Measuring Information Leakage

John Demme Robert Martin Adam Waksman Simha Sethumadhavan
Computer Architecture Security and Technology Lab
Department of Computer Science, Columbia University, NY, NY 10027
jdd@cs.columbia.edu, rdm2128@columbia.edu, {waksman,simha}@cs.columbia.edu

Abstract

There have been many attacks that exploit side-effects of program execution to expose secret information and many proposed countermeasures to protect against these attacks. However there is currently no systematic, holistic methodology for understanding information leakage. As a result, it is not well known how design decisions affect information leakage or the vulnerability of systems to side-channel attacks.

In this paper, we propose a metric for measuring information leakage called the Side-channel Vulnerability Factor (SVF). SVF is based on our observation that all side-channel attacks ranging from physical to microarchitectural to software rely on recognizing leaked execution patterns. SVF quantifies patterns in attackers' observations and measures their correlation to the victim's actual execution patterns and in doing so captures systems' vulnerability to side-channel attacks.

In a detailed case study of on-chip memory systems, SVF measurements help expose unexpected vulnerabilities in whole-system designs and shows how designers can make performance-security trade-offs. Thus, SVF provides a quantitative approach to secure computer architecture.

1 Introduction

Data such as user inputs tend to change the execution characteristics of applications; their cache, network, storage and other system interactions tend to be data-dependent. In a side-channel attack, an attacker is able to deduce secret information by observing these indirect effects on a system. For instance, in Figure 1 Alice uses a service hosted on a shared system. Her inputs to that program may include URLs she is requesting, or sensitive information like encryption keys for an HTTPS connection. Even if the shared system is secure enough that attackers cannot directly read Alice's inputs, they can observe and leverage the inputs' indirect effects on the system which leave unique signatures.

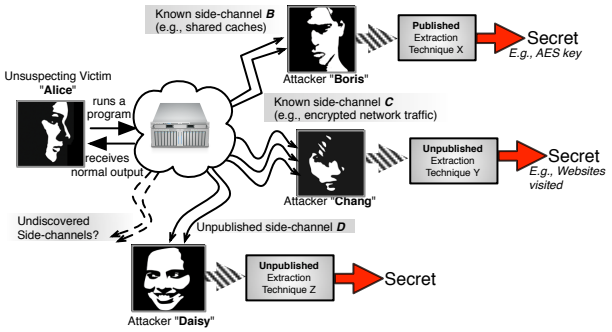
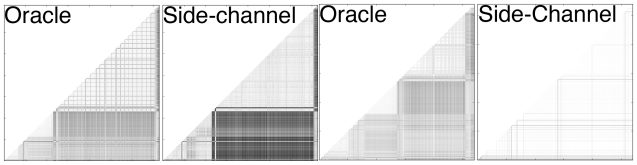


Figure 1: Information leaks occur as a result of normal program execution. Alice's actions can result in side effects. Attackers can measure these side effects as "side-channel" information and use it to extract secrets using known or unpublished attack techniques.

For instance, web pages have different sizes and fetch latencies. Different bits in the encryption key affect processor cache and core usage in different ways. All of these network and processor effects can and have been measured by attackers. Through complex post-processing, attackers are able to gain a surprising amount of information from this data.

While defenses to many side-channels have been proposed, currently no metrics exist to quantitatively capture the vulnerability of a system to side-channel attacks.

Existing security analyses offers only existence proofs that a specific attack on a particular system is possible or that it can be defeated. As a result, it is largely unknown what level of protection (or conversely, vulnerability) modern computing systems provide. Does turning off simultaneous multi-threading or partitioning the caches truly plug the information leaks? Does a particular network feature obscure information needed by an attacker? Although each of these modifications can be tested easily enough and they are likely to defeat existing, documented attacks, it is extremely difficult to show that they increase resiliency to future attacks or even that they increase difficulty for the



(a) A high SVF system ($\approx$0.77) (b) A low SVF system ($\approx$0.098)

Figure 2: Visualization of execution patterns. LHS of each figure shows "ground-truth" execution patterns of the victim and RHS shows patterns observed by the attacker for two different microarchitectures. One can visually tell that the higher SVF system (left) leaks more information.

attacker using novel improvements to known attacks. To solve this problem, we present a quantitative metric for measuring side channel vulnerability.

We observe a commonality in all side-channel attacks: the attacker always uses patterns in the victims program behavior to carry out the attack. These patterns arise from the structure of programs used, typical user behavior, user inputs, and their interaction with the computing environment. For instance, memory access patterns in OpenSSL (a commonly used crypto library) have been used to deduce secret encryption keys [13]. These accesses were indirectly observed through a shared cache between the victim and the attacker process. As another example, crypto keys on smart cards have been compromised by measuring power consumption patterns arising from repeating crypto operations [9].

In addition to being central to side channels, patterns have the useful property of being computationally recognizable. In fact, pattern recognition in the form of phase detection [5, 14] is well known and used in computer architecture. In light of our observation about patterns, it seems obvious that side-channel attackers actually do no more than recognize execution phase shifts over time in victim applications. In the case of encryption, computing with a 1 bit from the key is one phase, whereas computing with a 0 bit is another. By detecting shifts from one phase to the other, an attacker can reconstruct the original key [13, 4]. Even HTTPS side-channel attacks work similarly – the attacker detects the network phase transitions from "request" to "waiting" to "transferring" and times each phase. The timing of each phase is, in many cases, sufficient to identify a surprising amount and variety of information about the request and user session [1]. Given this commonality of side-channel attacks, our key insight is that side-channel information leakage can be characterized entirely by recognition of patterns through the channel. Figure 2 shows an example of pattern leakage through two microarchitectures, one of which transmits patterns readily and one of which does not.

	Example Insecure CPU	Example Secure CPU	Similar to attacked CPUs in [13], [4]	
SVF	0.86	0.01	0.73	0.27
SMT Cache Sharing	2-way L1	1-way L2	2-way L1	2-way L1
L1D Size	1k	32k	8k	32k
L1D Associativity	4-way	4-way	4-way	8-way
L1D Line Size	8B	64B	64B	64B
L1D Prefetcher	Arithmetic	None	None	None
L1D Partitioning	Static	Static	None	None
L1D Latency	4 cycles	4 cycles	2 cycles	3 cycles
L2 Size	8k	256k	512k	1M
L2 Associativity	4-way	4-way	8-way	8-way
L2 Line Size	8B	8B	64B	64B
L2 Prefetcher	None	None	Arithmetic	Arithmetic
L2 Partitioning	Static	Static	None	None
L2 Latency	16 cycles	16 cycles	10 cycles	7 cycles

Table 1: SVFs of example systems running an OpenSSL RSA signing operation. We have selected two hypothetical systems from our case study in addition to approximations of processors which have been attacked in previous cache side-channel papers [4, 13]. It is interesting to note that while the processor with an SVF of 0.27 was vulnerable to attack, the attack required a trained artificial neural network to filter noise from the attacker observations. The 0.73 SVF system required no such filtering to be attacked.

Accordingly, we can measure information leakage by computing the correlation between ground-truth patterns and attacker observed patterns. We call this correlation Side-channel Vulnerability Factor (SVF). SVF measures the signal-to-noise ratio in an attacker's observations. While *any* amount of leakage could compromise a system, a low signal-to-noise ratio means that the attacker must either make do with inaccurate results (and thus make many observations to create an accurate result) or become much more intelligent about recovering the original signal. This assertion is supported by published attacks given in Table 1. While the attack [13] on the 0.73 SVF system was relatively simple, the less vulnerable 0.27 system's attack [4] required a trained artificial neural network to filter noisy observations.

As a case study to demonstrate the utility of SVF, we examine the side-channel vulnerability of processor caches, structures previously shown to be vulnerable [12, 11, 13, 4]. Our case study shows that design features can interact and affect system leakage in odd, non-linear manners. Evaluation of a system's security, therefore, must take into account all design features. Further, we observe that features designed or thought to protect against side-channels (such as cache partitioning) can themselves leak information. Our results show that predicting vulnerability is difficult, therefore it is important to use a quantitative metric like SVF to evaluate whole-system vulnerabilty.

To summarize, we propose a metric and methodology for measuring information leakage in systems; this metric represents a step in the direction of a quantitative approach to whole system security, a direction that has not been ex-

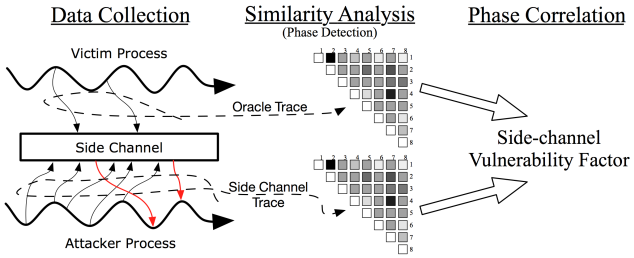


Figure 3: An overview of computing SVF: two traces are collected, then analyzed independently for patterns via similarity analysis, and finally correlation between the similarity matrices is computed.

plored before. We also evaluate cache design parameters for their effect on side-channel vulnerability and present several surprising results, motivating the use of a quantitative approach to security evaluation.

The rest of the paper is organized as follows: Section 2 describes the method used for calculating SVF in general then Section 3 discusses SVF's caveats and several limitations. The rest of the paper then illustrates how SVF can be applied to on-chip memory systems. Section 4 describes our evaluation methodology and Section 5 presents SVF analysis for the on-chip memory system. Section 6 discusses related work. Finally, Section 7 presents conclusions and future work.

2 Side-channel Vulnerability Factor

Side-channel Vulnerability Factor (SVF) measures information leakage through a side-channel by examining the correlation between a victim's execution and an attacker's observations. Unfortunately these two data sets cannot be directly compared since they represent different things – for instance, instructions executed versus power consumed. Instead, we use phase detection techniques to find patterns in both sets of data then compute the correlation between actual patterns in the victim and observed patterns.

An overview of SVF computation is shown in Figure 3. We begin by collecting oracle and side-channel traces. These traces are time-series data which represent the important information an attacker is trying to observe and the measurements an attacker is able to make, respectively. We then build similarity matrices for each trace and compute the correlation between these two matrices.

2.1 System Specification

Oracle The oracle trace contains ground-truth about the execution of the victim. It is the information which an attacker is attempting to read; in an ideally leaky side-channel, the attacker could directly read the oracle trace. For instance, one might use memory references (as we do in the upcoming case study) so the resulting SVF would indicate how well memory reference patterns are observed through the side-channel.

Side-Channel The side-channel trace contains information about events which the attacker observes. The side-channel data should be realistic with respect to data which an attacker can practically measure. For instance, in an analog attack, the side-channel trace may be instantaneous power usage over time. In a cache side-channel, the trace may be the latency to access each cache line over time.

Distances For each trace, we will be detecting execution phases. This involves comparing parts of each trace to every other part. This comparison is done simply with a distance function, the selection of which will depend on the data types in the two traces. For instance, if they are represented as vectors, one might use Euclidean distance. If a trace contains bits from an encryption key, the distance function may simply be equality.

2.2 Similarity Matrix

After collecting oracle and side-channel traces, we have two series of data. However, the type of information in each trace is different. For instance, the attacker may measure processor energy usage during each time step whereas the oracle trace captures memory accesses in each time step. As a result, we cannot directly compare the traces. Instead, we look for patterns in each trace by computing a similarity matrix for each trace. This matrix compares each time step to every other in the sequence. For a sequence S of length $|S|$, we define each element of the similarity matrix M of size $|S|^2$ as:

$$M(i,j) = \begin{cases} D(S_i, S_j), & \text{if } i > j \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where D is a distance function between two time steps. This definition creates a triangular matrix without the diagonal. In all cases (due to the definition of a distance function) the diagonal will contain all zeros in both the oracle and side-channel matrix. So, if we included the diagonal in the matrix, we would observe a false (or trivial) correlation.

2.3 Correlation

In the previous step, we build similarity matrices for both the oracle and side-channel information. Patterns in execution behavior present in the original traces are reflected in these matrices. Indeed, these patterns are often visually detectable (see Figure 2a). A maximally leaky side-channel will faithfully mirror the patterns in the oracle trace. However, if the side-channel conveys information poorly, it will be more difficult to discern the original pattern, as in Figure 2b. We can determine presence of pattern leakage by computing the correlation between the two matrices. Specifically, for each element in the oracle matrix, we pair it

with the corresponding element in the side-channel matrix. Given this list of pairs, we compute the Pearson correlation coefficient, more commonly known simply as correlation. The closer this value is to one (it will never be above one), the more accurately an attacker is able to detect patterns. The closer the coefficient is to zero, the less accurately the attacker is observing patterns. At values very near zero, the attacker is essentially observing noise and we are measuring random correlation.

After working through all these steps, we call the final correlation the Side-channel Vulnerability Factor.

3 Applications, Caveats and Limitations

Now that we have defined SVF, we review some caveats which must be considered when using SVF and finally discuss some of the limitations of SVF.

3.1 Caveats

SVF analysis relies on a number of assumptions that are reasonable based on known facts about attack models and intuition about program and system behavior. However it is possible for situations to exist in which SVF may not capture vulnerability clearly. For uses beyond the case study presented in remaining sections, the following issues must be evaluated and small changes to SVF may be necessary.

Self-Similarity Pattern Analysis SVF analysis assumes that attackers use time-varying information (*e.g.,* changes in cache miss rates over time) and look for patterns in the changes. If an attacker is able to gain information from single measurements (without looking at changes over time), SVF cannot capture such measurements. For instance, if an attacker is able to measure the number of evictions in some cache sets, make these measurements only *once* and gain sensitive information, then SVF will not detect the leak. However, these leaks are often easily defended; in this example, randomized hashing would likely be effective.

Linear Correlation SVF analysis as presented here compares oracle and side-channel similarity matrices using the Pearson correlation coefficient. This correlation test is only robust to linear correlation. If some non-linear correlation is expected then the SVF analysis should use alternate correlation metrics.

Latency Effects SVF computes correlation between time-aligned elements of the similarity matrices. This implicitly assumes that the attacker receives information through the side-channel instantly. If the attacker's information from the victim is delayed the correlation computation must be adjusted for the latency, for instance using cross-correlation.

3.2 Limitations

Known side-channels Since SVF requires the definition of a side-channel trace, it can be computed only for known (or suspected) side-channels. If an attacker has discovered an unknown side-channel and is secretly using it, SVF cannot be used to compute the vulnerability of the system to this secret side-channel. However, SVF can be used to help find new side-channels. For instance, in our later case study of caches, we quantitatively discover a side-channel through a pipeline since it interacts with the cache side-channel.

Relativity SVF is a relative metric. For two systems A and B, if the SVF of A is greater than SVF B, SVF only says that A leaks more than B. It does not translate to ease of hacking or hacker hours to carry out an attack. This translation requires a model of human creativity, knowledge, productivity etc., which is likely impossible. SVF also is a function of the side-channel trace, so an attacker is implicitly defined. To create an absolute metric completely independent of an attack we need to measure to total amount of information leaked during a computation. Currently we do not know how to do this. It is also likely that this bound will be too high to be practically useful; since we know that computations are orders of magnitude less energy efficient than theoretically possible, they likely leak much information. For this reason, we use a relative metric, just as we do when reporting energy efficiency.

4 Case Study: Memory Side-Channels

Since shared cache microarchitectures have been exploited in the past [12, 11, 13, 4], it is important to characterize the vulnerability of cache design space. Are there cache features which obscure side-channels? Do protection features reduce vulnerability to undiscovered attacks in addition to known attacks? To answer these questions, we simulate 34,020 possible microarchitectural configurations running OpenSSL RSA algorithm and measure their SVF. In this section, we describe our methodology for computing SVFs. The next section presents the results.

4.1 Framework for Understanding Cache Attacks

A cache side-channel attack can exist whenever components of a chip's memory are shared between a victim and an attacker. Figure 4 illustrates a typical cache attack called the "prime and probe" attack. In this attack style, the attacker executes cache scans during the execution of a victim process. During each cache scan, the attacker scans each set of the cache by issuing loads to each way in the set and measuring the amount of time to complete all the loads. If the loads complete quickly, it is because they hit in the cache from the last cache scan. This further implies that there is no contention between the victim and attacker in this cache set. Inversely, if the loads are slow, the attacker assumes contention for that set. By measuring the contention for many cache sets, an attacker obtains an image of the victim's working set. Further, by regularly remeasuring this contention, the attacker measures shifts in

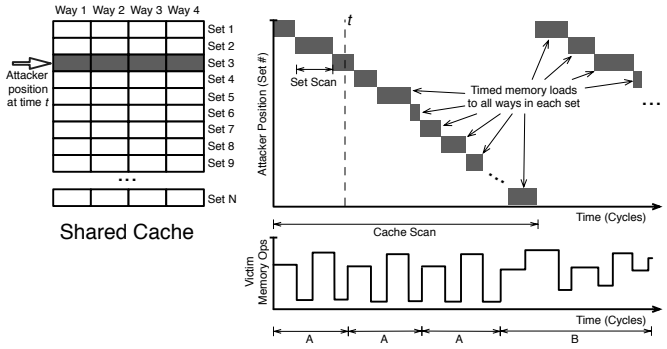


Figure 4: In cache side-channel, an attacker times loads to each set in a shared cache one set at a time. The amount of time to complete the loads indicates the amount of contention in the set, thus leaking data about a victim's working set.

the victim's working set. These shifts represent phase shifts in the victim and often implicitly encode sensitive information.

In the example of Figure 4, the victim repeats a distinct memory access pattern A. This repetition cannot be detected by the attacker because the pattern is much shorter than the scan time. The victim's shift from access pattern A to pattern B, however, can likely be detected. In general, caches with fewer sets allow attackers to scan faster and thus detect finer granularity behavior. However, smaller caches divulge less information about the victim's behavior during by each scan. It is intuitively unclear which factor is more important, though our results in the next section imply that speed is the critical factor.

There can also exist cases where the victim applications' important phase shifts occur more slowly than in our example. Further, they may occur much more slowly than the attacker's cache scans. In this case, it may make sense for an attacker to combine the results of multiple cache scans (by adding them), hopefully smoothing out any noise from individual scans. We call one or more scans an interval, and it defines the granularity of behavior which an attacker is attempting to observe. We characterize the length of these intervals by calculating the average number of instructions which the victim executes during each interval.

4.2 SVF Computation Specifications

Based on this outline for cache side-channel attackers, we define traces for our oracle (the information an attacker is trying to obtain) and side-channel (information the attacker can obtain) and methods of comparing them, as outlined in Section 2.1.

Side-channel As demonstrated in Figure 4, our attacker scans each cache set and records a time. Each time the attacker completes a scan through the entire cache, it assembles a vector of the measured load times for each set. We can then compare the results of a cache scan to any other cache scan using Euclidean distance. This distance gives the attacker a measure of the difference between the victim's working sets at the times when the two scans were taking place.

Oracle Attackers execute cache scans in an attempt determine a victim's working set. In order to determine how accurately the attacker obtains this information, we must measure an oracle of the victim's working set. In simulation this is easily obtained by recording the memory locations touched by the victim during each attacker cache scan. We build a vector of the number of accesses to each memory location during each cache scan. While these vectors cannot be directly compared to the vectors obtained from the attackers, they can be compared against each other using Euclidean distance to obtain distances between actual working sets when attacker cache scans were taking place.

Correlation Optimization We compute SVF for an entire execution of OpenSSL which will have many intervals. Unfortunately, computing the similarity matrices described in Section 2.2 requires quadratic time and space with the number of intervals. To avoid this problem, we instead use a random subset of the matrix proportional to the number of intervals. We have found this to be an accurate approximation of the full computation.

4.3 Attacker Capabilities

We model six different types of attackers representing different cache scan patterns and abilities to circumvent microarchitectural interference.

A simple attacker will likely scan each cache set in order. However, there are two other options. A random permutation of this ordering (determined before execution and held constant) may yield different information and may also assist in avoid noise due to prefetching. Second, it may be that an attacker can obtain a sufficient snapshot from only a small portion of the cache sets. In our random subsets attacker, we randomly select 25% of the cache sets and scan only those sets, decreasing the cache scan time by 4x.

It is also likely that complex prefetching techniques add noise to an attacker's observations. However, if an attacker has enough knowledge about the prefetcher, it may be able to effectively disable or otherwise negate these effects. As such, we model attacks with prefetching enabled and also attacks when prefetching is disabled on the attack thread, simulating a "prefetch-sensitive" attacker.

4.4 Microprocessor Assumptions

We model a microprocessor with two integer execution units, two floating point/SSE units and a single load/store unit. Up to three loads or stores can be issued each cycle from a 36-entry dispatch stage; the load store buffer has 48 load and 32 store entries. The load/store unit im-

Replacement Policy	LRU
L1 Latency	4 cycles
L2 Latency	16 cycles
L3 Latency	54 to 72 cycles
Inter-cache B/W	16 bytes / cycle
Cache Request Buffer	16 entry
Outstanding Misses (all levels)	4

Table 2: Fixed cache design parameters

plements store forwarding and redundant load coalescing. The branch predictor we model is a simple two-level predictor. Since we are focusing on the cache hierarchy, we assume perfect memory disambiguation, perfect branch target prediction, and no branch misprediction side effects. Using these perfect microarchitectural structures reduces execution variability and thus likely improves the quality of side-channel information, increasing the SVF of a particular system. Thus our simulation results should be somewhat conservative.

4.5 Experimental Parameters

We are interested in understanding the impact of cache size, line size, set associativity, hashing function, prefetchers, and several side-channel countermeasures such as partitioning schemes and eviction randomization. Details of our design space parameters follow:

Cache Size We simulate 1KB, 8KB and 32KB L1D cache sizes. The L1I, L2, and L3 caches are sized relative to the L1D cache at ratios of 1x, 8x, and 256x respectively. As demonstrated in Figure 4, larger caches will take longer to scan. However, the amount of information obtained by each scan will be greater.

Line Size We study line sizes (in all cache levels) of 8 and 64 bytes. Small line sizes increase the resolution of side-channel information – the attacker can get more precise information about victim addresses – but requires more sets to get the same cache size, thus increasing the amount of time it takes to scan the cache.

Set Associativity In a fully associative cache an attacker cannot get any information about the addresses a victim is accessing; it can do no better than determine how much overall contention there is. A direct-mapped cache, however, gives the attacker information about the victim's usage of each cache line. We would, therefore, expect varying the set associativity to affect information leakage. We study 1, 4, and 8 way caches with the set associativity identical for all levels.

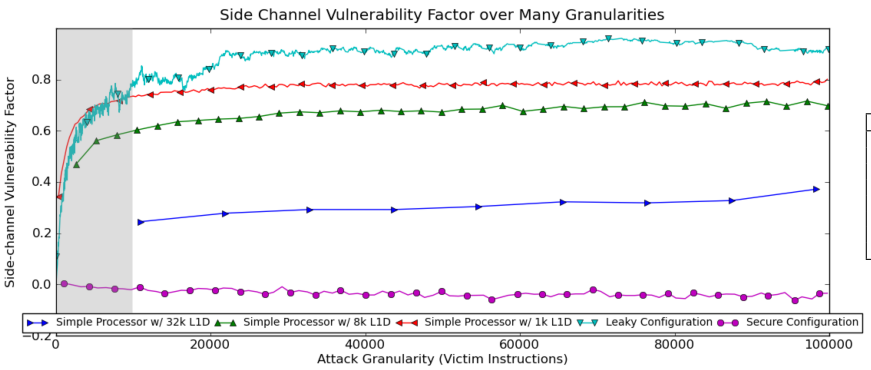
Hashing We study three hashing schemes for indexing cache sets. The first is the simplest, which is to index by the low order bits of the address. The second is a bitwise XOR of half of the bits with the other half, which is another common technique. We also study permutation register sets (PRS), a mechanism proposed for the RPCache scheme from Wang and Lee [16]. PRS maintain a permutation of the sets in the cache. We adapt PRS to our simulator with the following specifications: once every 100 loads, we change the permutation by swapping the mapping of two randomly selected sets. We also maintain different PRS for each thread, so the attacker and victim's mappings to cache sets are different. As a result of changes to the set mapping (which results in evictions), OpenSSL experiences an average slowdown of 6%. Although our implementation is different from Wang and Lee [16] to account for SMT we obtain similar results.

Prefetching Prefetchers may create noise in the side-channel as they initiate loads that (from the attacker's perspective) pollute the cache with accesses which the victim did not directly initiate. Conversely, prefetchers are essentially doing pattern detection, so it may be that they are able to amplify the effects of these victim memory patterns by prefetching based on those patterns. In addition to no profetching, we evaulated four prefetchers: next line, arithmetic [3], GHB/PCCS, and GHB/PCDC [10]. The "next" prefetcher is always used in the L1I cache. No prefetching is used at the L2 or L3 level and only one is turned on at once in the L1D.

Partitioning We would expect partitioning to reduce the amount of information an attacker obtains since it disallows direct access to a subset of the cache's lines. We use three policies to balance thread usage in shared caches. The first policy is to have no explicit management. The second is a static partitioning assignment – each process gets half of the ways in each set, however the cache load miss buffers and ports are shared. The last policy is a simple dynamic partitioning scheme. This scheme tracks per-thread usage in each set. Once every 10^6 cycles, the ways of each set are re-allocated between threads. If one thread is using a set more than twice as often as the other thread it is allocated 75% of the ways for the next 10^6 cycles.

Eviction Randomization A simple method of obscuring side channel information is random eviction. This introduces noise into the side-channel. We implement a policy that randomly selects a cache line and evicts it. This randomized eviction is activated either never, every cycle with 50% probability or every cycle.

SMT Finally, we are interested in studying how much simultaneous multithreading (SMT) contributes to information leakage. SMT potentially introduces a side-channel via the pipeline as contention for resources like the load/store queue and functional units could allow an attacker to sense a victim's activity based on interference in these units. Consequently, one would expect SMT configurations to yield more side-channel information. SMT-based attacks, how-



	Simple	Leaky	Secure
SMT	On	Off	Off
Cache Sharing	L1	L1	L2
L1D Size	32k/8k/1k	1k	1k
Line Size	64	64	64
Ways	8	8	4
Attacker	In Order	Subset	In Order
L1 Prefetcher	None	Next Line	None
Partitioning	None	None	Static

Figure 5: SVF for several memory system configurations executing OpenSSL's RSA signing algorithm over a range of attack granularities. We see that memory subsystem significantly impacts on the quality of information an attacker can obtain. Note that "leaky" and "secure" are *not* the *most* leaky or secure, merely two configurations towards each end of the spectrum. Leaky represents L1 cache sharing without SMT in the style of core fusion or composable processors.

ever, are easily foiled by simply disabling SMT or disallowing SMT sharing between untrusted processes. To model this "protected" configuration, we also simulate the victim and attacker threads running simultaneously on different cores (so they share no pipeline resources) wherein the cores share caches at either the L1 or L2 level. Although the shared L1 configuration is rare, this configuration allows us to directly compare against SMT configurations to determine the extent to which pipeline side-channels contribute to SVF.

Some of the cache design features are fixed, limiting our design space. For instance, prefetch requests are only considered if there is space available in the request buffer; we also do not modify prefetcher aggressiveness. We do not model OS interference like process swapping or interrupts as these effects are likely to add noise to the side-channel so removing them strengthens the attacker. Other fixed design choices are shown in Table 2.

5 Results

Our results are drawn from simulations of possible configurations varying core configuration (SMT, cache sharing), cache size, line size, set associativity, hashing function, prefetcher, partitioning scheme and eviction randomization policies. Here we present results about the impact of each of these factors on SVF.

5.1 Interval Sizing

As discussed in the last section and Figure 4, an attacker may combine multiple cache scans into an interval. This combining may help smooth out noise, so information gathered about the victim over larger time spans may be more accurate. Ideally, the intervals are sized to align with the natual phases of the victim. There are many possible interval sizes, and choosing an effective one depends on various system parameters as well as characteristics of the victim application. Instead of computing the SVF for a particular interval size, we do so for a large range of them beginning with the finest possible, the time it takes for the attacker to complete one scan of the all the cache lines. Figure 5 shows a graph of many SVFs over a wide range of interval sizes for several cache implementations. There are several interesting conclusions we can draw from this graph.

1. The SVFs for these systems range widely from essentially zero to nearly one. This means that configurations exist with virtually no potential for cache leakage (small absolute values of SVFs indicate essentially no leakage) while others leak heavily.

2. In the 32k L1D cache configuration, in the time it takes for the attacker to scan the cache once, about 11,000 victim instructions have committed on average, thus its line begins at 11,000 on the X axis. As a result, the inorder attacker cannot gather side channel information leaked during 11,000 instructions. The 8k L1D and 1k L1D caches, on the other hand, can be scanned much more quickly than the 32k L1D cache and (as a result) much more information is obtained.

3. SVF tends to increase with interval size. This intuitively makes sense; one would expect it to be easier to get accurate information about larger time spans than shorter ones.

4. Despite a general trend upward SVF can vary widely, indicating that (for an attacker) interval size selection is important. These peaks and valleys likely indicate areas where the attacker's interval size aligns with phase shifts in the victim application.

Notes on Data Analysis and Presentation For the rest of the paper, due to space considerations, we present only a subset of the intervals sizes shown in Figure 5. Specifically, we examine the case of fine granularities — which could be used to recover information like encryption bits — which we define as less then 10,000 committed victim memory instructions. This is represented by the shaded region in Figure 5. For each memory system configuration, we will use the maximum SVF observed in this region, as this represents an attacker which has selected an optimal interval size.

In order to aggregate the data from many simulations in a meaningful way, we present cumulative distribution function (CDF) plots of number of microarchitectural instances that have a value less than a given SVF. In each diagram, each line represents a large set of microarchitecture implementations. Sets which are more secure will have lines to the left of less secure sets. This format allows us to answer many different questions. For instance, does a particular feature allow us to close a leak entirely? For how many configurations does it do so? For example, in Figure 6 we see that in this set of configurations, turning off SMT results in lower SVF. However, when SMT is turned off, not sharing the L1 is usually more secure, but not in all cases.

5.2 Core Configuration

Several side channel attacks take advantage of SMT capabilities. Accordingly, a first reaction to defeat these attacks is to simply turn off SMT. Does this indeed eliminate cache side channels? Figure 6 demonstrates that it does not, though it helps. While SMT (which implies a shared L1D) provides more information to the attacker than no SMT (which realistically means the L1D is not shared), there are still many configurations in which an attacker gets information through sharing in the L2.

To evaluate the leakiness of the pipeline, we also include a relatively unrealistic configuration which turns off SMT but still shares an L1 (*e.g.,* Core Fusion composition). These data indicate that the pipeline side channel offers additional information to the attacker, even if the pipeline is not specifically targeted by the attacker. It is thus likely that existing SMT-based attacks implicitly benefit from pipeline side channel information in addition to cache side channels.

5.3 Cache Design Space Exploration

Caches come in many different flavors; sizes, set associativity, prefetchers and other features differ amongst designs. Additionally, cache designers may implement side channel protection features such as randomized eviction, randomized hashing or cache partitioning. In this section, we examine the effect that some implementations of these features can have on the cache's vulnerability. Figure 7 shows these results in an SMT system, so in all cases both the L1 caches and pipeline are shared between the attacker and victim. The results we have obtained are specific to our simulation model, workload choice and attacker implementations.

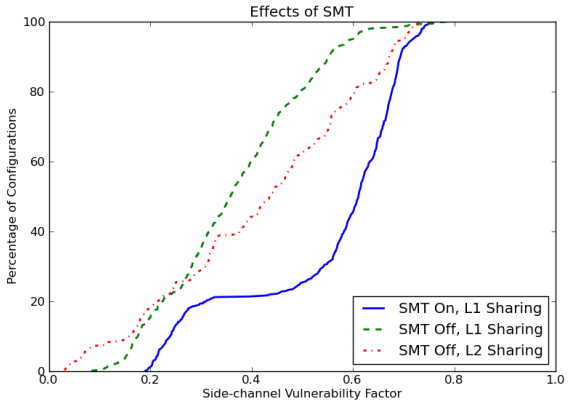


Figure 6: This cumulative histogram shows the percentage of configurations with various sharing configurations which have SVFs no more than the value on the X axis. For instance, all SMT configurations have SVFs of at least 0.2 and only 40% of them have SFVs less than 0.6. Without SMT, however, the SVF *can* be reduced to nearly zero, but many non-SMT configurations still leak information. Note that none of the configurations represented here have protection mechanisms like cache partitioning.

Cache Size One of the largest determinants of SVF is cache size. Consistent with the data in Figure 5, the cache size graph in Figure 7 indicates that larger caches leak less. This can be attributed to the time it takes for the attacker to scan the cache. Larger caches take longer to scan, so in the time it takes an attacker to scan the cache, the victim makes much more progress and thus the attacker misses fine grained OpenSSL behaviors.

Line Size We expected that the smaller the line size, the more resolution an attacker can obtain about addresses being accessed. The next graph, however, shows that line size selection does not seem to make a huge difference to cache vulnerability.

Associativity The set associativity graph contains some interesting results. We see that increased associativity provides the opportunity for decreased vulnerability, though not in all situations. There are likely two reasons for this. First, more ways in a set decreases the precision of the side channel; missing in an 8-way set tells an attacker that one of the 8 locations the attacker pre-loaded was evicted whereas in a direct mapped cache, a miss tells the attacker with certainty about a particular line. However, increasing the number of ways slightly increases the speed at which the attacker can scan (since OoO cores allow the loads in each set scan to execute in parallel) and speed is an important factor.

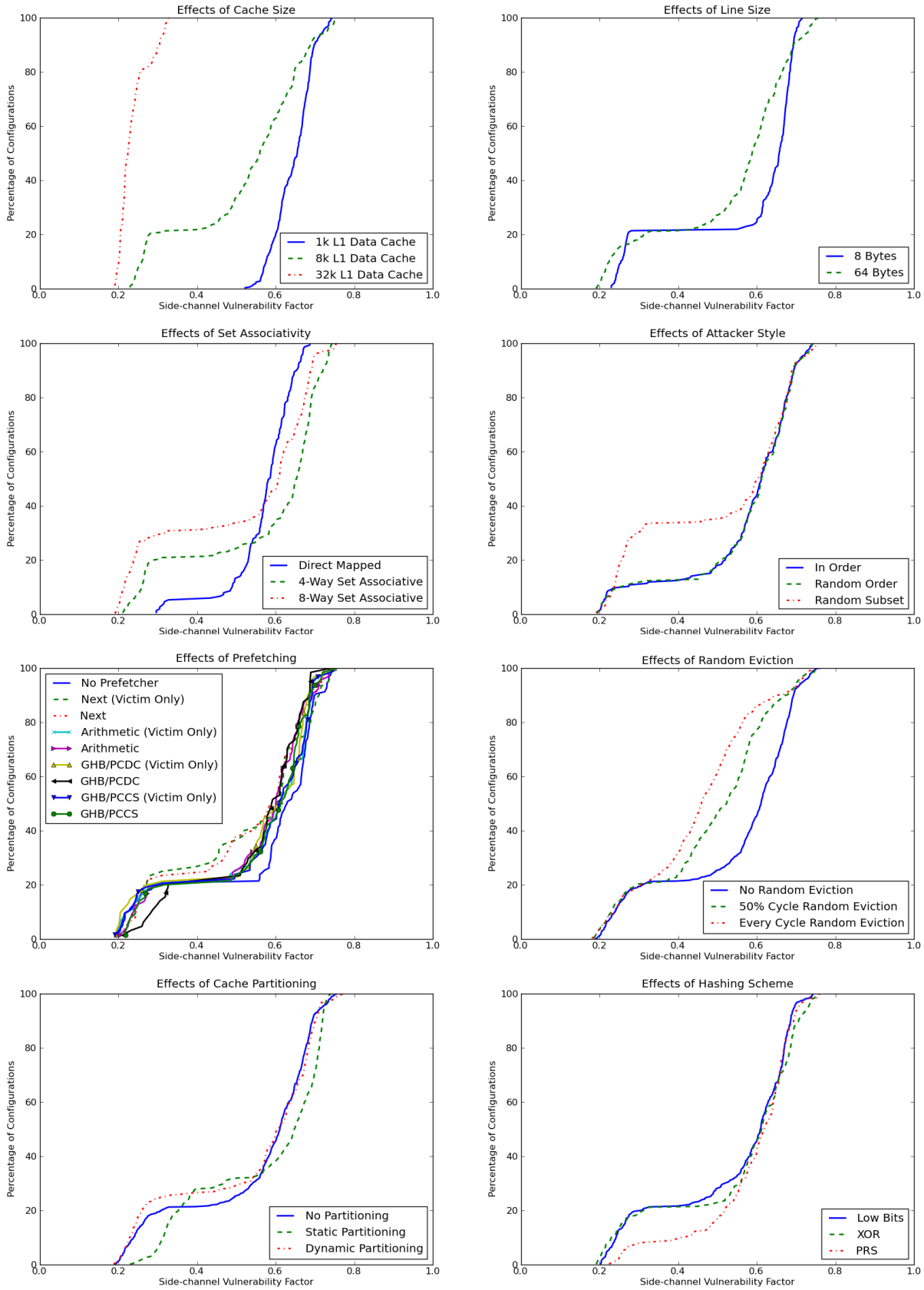


Figure 7: Cumulative histograms indicating the small interval size vulnerability of various cache features. In all but the last two cases, configurations are limited to SMT on (with shared level 1) with no protection (like partitioning and random eviction). In each graph, a set of features are selected and we draw a cumulative histogram with respect to Side-channel Vulnerability Factor. In short, lines (features) to the left of others have more configurations with a lower SVF – a desirable trait.

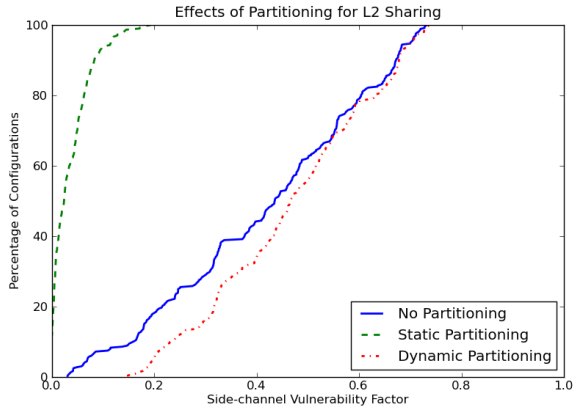


Figure 8: Static partitioning is extremely effective when SMT is turned off. Since this disables pipeline side channels and static partitioning disables cache *content* side channels the only remaining side channels are shared cache buffers and ports, which are not terribly effective side channels. Additionally, we see that a simple dynamic partitioning mechanism can *itself* leak information.

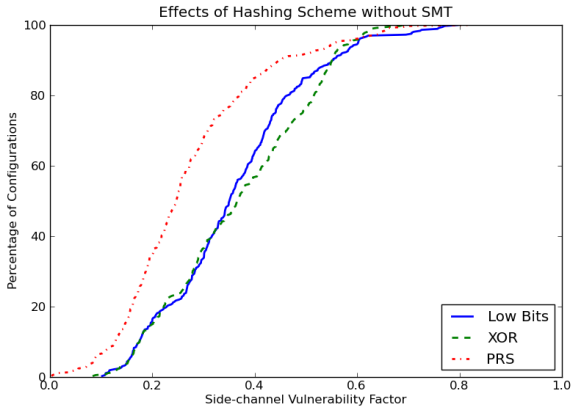


Figure 9: In some SMT configurations, our implementation of permutation register sets (PRS) can leak information. However, this is largely because it does not address pipeline side channels, yet slows down the victim. If we turn off SMT yet still share the L1 cache (as in this figure), we see that PRS obscures the side channel as expected.

Attacker Style Three different attacks have been tested. Two of them – inorder and random order – are nearly identical; they differ only in their ordering of the cache sets during their scans, so nearly the same information is obtained from both, though at slightly different times. The random subset attack, however, is substantially different as it scans only 25% of the cache. These data imply that in about 70% of cache configurations (10% with low SVFs, 60% with high SVFs) examining less data but doing so 4x faster is a fair trade off. In the remaining 30%, however, the attacker misses critical data.

Prefetching We expected prefetching to significantly degrade information leakage as it often accurately predicts and prefetches cache lines which the attacker would have otherwise missed. These data, however, contradict this intuition. In some cases, we assume that the attacker can defeat the prefetcher (effectively turning it off) and in others we assume that it cannot. In both cases we see that prefetching does not heavily degrade the side channel. This is likely because prefetchers are deterministic and guided by address streams; we can think of them as a deterministic transform on the pattern rather than information destruction.

Random Eviction One might expect randomly evicting cache lines to introduce noise, and thus degrade the side-channel. Our simulations indicate that this is true, but only to a relatively small extent. Further, this technique is only effective on about 70% of cache configurations.

Partitioning Another protection mechanism is partitioning. Partitioning protects caches by sometimes disallowing the sharing interference which the attacker measures. As such, we would expect partitioning to degrade the side-channel. Our data, however, do not support this expectation. As we saw in Figure 6, other side channels exist in the shared system and even with partitioning, these other side channels can be exploited. In some cases static partitioning even strengthens the attacker. This can be explained by the fact that the attacker runs faster allowing other side channels to be polled much more often. This is not to say, however, that static partitioning is ineffective. Figure 8 shows the effectiveness of partitioning in systems without SMT and a shared L2. Although the dynamic partitioning mechanism *itself* leaks information, static partitioning is a very effective protection mechanism.

Hashing Scheme Lastly, we look at the effect of hashing schemes. We see that there is virtually no difference between using the low bits of an address and XOR'ing parts of it. This is to be expected because XOR'ing amounts to relatively simple reordering of cache lines rather than information loss. Our implementation of permutation register sets, however, ends up slowing down the victim (about 3% on average, more for some configurations) and thus the timing and pipeline channels are able to get more information. Kong *et al.* [8] also find vulnerabilities in RPCache, supporting our results. However, PRS is not always more leaky; in Figure 9 we look at cache configurations with SMT turned off and see that in this case, PRS helps obscure the side-channel information. In other words, PRS performs exactly as expected: it protects against the *cache line sharing* side-channel. In doing so, however, it can make victims more vulnerable to other side-channels like a shared pipeline channel.

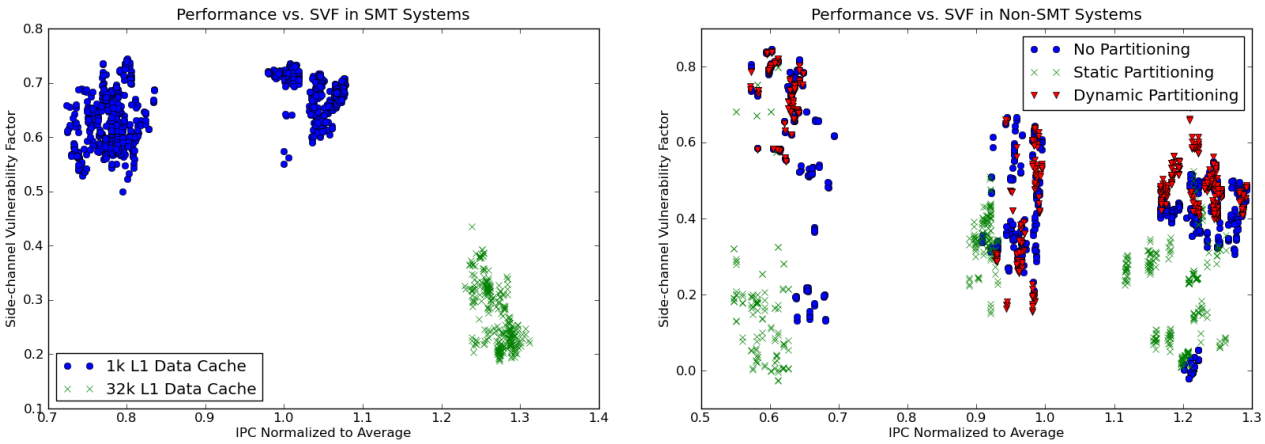


Figure 10: Many security proposals result in decreased performance. However, these results indicate that this need not always be the case. For instance, in SMT processors (on the left) increasing the cache size both decreases SVF and increases performance. In non-SMT systems (on the right) there exist high-performance systems with very low SVFs using both no cache partitioning and static partitioning.

5.4 Performance and Security

Some protection features may incur a performance penalty. For instance, both PRS and random eviction often make systems more secure but both methods degrade performance. In Figure 10 we examine performance tradeoffs in SMT and non-SMT processors. In the SMT case, we see that SVF decreases are correlated with performance improvements. In the non-SMT case, there is no correlation but there exist configurations with both low SVF and good performance. Further, we also observed in the last subsection that faster victim execution often means better security as it is harder to observe a moving target. Our conclusion, therefore, is that performance and security need not always be traded off.

5.5 Broader Observations

Through simulation and SVF computation, we have examined the effect of cache design choices on the cache's vulnerability to side channels. We must stress that the results of our case study are specific to our simulation model: we cannot and do not claim these results to generalize to other models or real processors. We recommend that microprocessor designers adapt SVF evaluation methodology to their simulation environments to obtain results for their specific designs.

However, there are several generalized lessons learned: (**1**) Any shared structure can leak information. Even structures intended to protect against side channel leakage can increase leakage. (**2**) No single cache design choice makes a cache absolutely secure or completely vulnerable. Although some choices have larger effects than others, several security-conscious design choices are required to create a secure shared system. (**3**) The leakiness of caches is *not* a linear combination of design choices. Some features leak information in some configurations but protect against it in others. Others only offer effective protection in certain situations. Predicting this leakiness is, therefore, extremely difficult and probably requires simulation and quantitative comparison like we have done in this paper.

6 Related Work

A side-channel is a method of gaining protected information that exploits the implementation of a system, rather than its theory or design. Side-channels can (and most likely will) exist in any given implementation and can be difficult to foresee, discover or protect against. Side channels can take many disparate forms including electrical signals, acoustic signals, microarchitectural and architectural effects, application level timing channels, and any other shared resource through which an eavesdropper can detect any information or state left by another program. Consequently there is a long history of side-channel attacks and countermeasures [7, 6, 16, 15, 8].

There has been little work on the science of side-channel security in form of experimental frameworks and metrics. To evaluate the security of caches, Dominitser *et al.* propose an analytical model to predict the amount of information leakage through cache side-channels [2]. The technique proposed in their paper tracks the fraction of the victim's critical items accessible in the cache to determine leakage. Our work differs in three aspects: first, our technique does not require data items to be marked as critical, secondly, as we have shown, focusing on caches alone is insufficient to evaluate side-channel leaks of cache based attacks. Finally, our metric can be used to determine leakage in any microarchitectural structure, and more broadly to full systems.

7 Conclusion

In this paper we introduced Side channel Vulnerability Factor (SVF), a metric intended to quantify the difficulty of exploiting a particular system to gain side channel information. Using SVF, we also presented the results of a study exploring the side-channel potential of a large cache design space. We find several surprising results, indicating that predicting the security of a system is extremely difficult; a quantitative, holistic metric is necessary.

As a result of using execution traces, SVF is useful beyond caches; one can compute SVF for any system for which oracle and side-channel traces can be defined. For instance, one could look at encryption keys on smart cards versus their power usage variability during encryption. SVF could also be used to find a correlation between an audio conversation and the size/rate of network packets observed by an intermediate node in the Skype network. Many systems lend themselves well to SVF analysis.

Another advantage of using execution traces is that they are often easily defined and measured. No mathematical modeling is required to compute SVF. This freedom may help discover or prevent new side channel leaks, as the same subtleties that allowed the leak to survive the design process may make accurate mathematical modeling difficult or impossible. Indeed, a recurring theme in the study of side-channel research is this: any shared structure can leak information. As such, only an end-to-end analysis, like SVF, which accounts for system level effects and oddities, can accurately determine side channel vulnerability.

Acknowledgements

We thank anonymous reviewers, Dr. Pradip Bose, and members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback on this work. This work was supported by grants FA 99500910389 (AFOSR), FA 865011C7190 (DARPA), FA 87501020253 (DARPA), CCF/TC 1054844 (NSF) and gifts from Microsoft Research, WindRiver Corp, Xilinx and Synopsys Inc. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities.

References

[1] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 191 –206, may 2010.

[2] L. Domnitser, N. Abu-Ghazaleh, and D. Ponomarev. A predictive model for cache-based side channels in multicore and multithreaded microprocessors. In *Proceedings of the 5th international conference on Mathematical methods, models and architectures for computer network security*, MMM-ACNS'10, pages 70–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] T. fu Chen and J. loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[4] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505, May 2011.

[5] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase shift detection: A problem classification, 2003.

[6] C. K. Koc. *Cryptographic Engineering*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[7] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[8] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, CSAW '08, pages 25–34, New York, NY, USA, 2008. ACM.

[9] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of power analysis attacks on smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*, WOST'99, pages 17–17, Berkeley, CA, USA, 1999. USENIX Association.

[10] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *Ieee Micro*, 25(1):90–97, 2004.

[11] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA*, pages 1–20, 2006.

[12] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30 – 44, 2003.

[13] C. Percival. Cache missing for fun and profit, 2005.

[14] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *Micro, IEEE*, 23(6):84 – 93, nov.-dec. 2003.

[15] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 83 –93, nov. 2008.

[16] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35:494–505, June 2007.