A Quantitative, Experimental Approach to Measuring Processor Side-Channel Security

Side-channel attackers infer secrets by observing unintentional side effects of program execution. The lack of an experimental methodology for understanding such side effects makes it difficult to address them in the early design stages. The proposed Side-channel Vulnerability Factor addresses this need, and is based on the observation that side-channel attacks rely on recognizing leaked execution patterns.

John Demme
Robert Martin
Adam Waksman
Simha Sethumadhavan
Columbia University

In a side-channel attack, an attacker can deduce secret information by observing indirect, unintentional effects that programs' inputs and execution have on a system. For instance, users often access services hosted on a shared system, and their inputs to the hosted program could include such sensitive information as requests for URLs or encryption keys for an HTTPS connection. Even if the shared system is secure enough that attackers can't directly read the users' inputs, they can observe and leverage the unique signatures of the inputs' indirect effects on the system to infer what the inputs were. For instance, different webpages have different sizes and fetch latencies, which affect network bandwidth, and different bits in an 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 can gain a surprising amount of information from such data.

Side-channel exploits affect a variety of important domains. Several attacks on shared cloud services have already been published,[1] and such shared systems will likely be attacked more in the future, requiring cloud operators to consider side-channel security. At the other end of the spectrum, mobile phones and other embedded systems—which are used for everything from authentication to accounting to entertainment systems—hold a lot of sensitive information and are also vulnerable to physical side-channel attacks. Smart cards, for instance, have had their encryption keys stolen through power-based side channels.[2] Securing these systems against side-channel attacks is now of interest to both industry and academia.

Documented side-channel attacks offer existence proofs of exploitable vulnerabilities, but they should also be considered as symptoms of a deeper problem. While most attacks demonstrated on real systems exploit leakage through shared caches,[1,3-5] other

Published by the IEEE Computer Society

Related Work on Reasoning About Information Leakage

A classic (blue sky) goal in information-flow security is noninterference.[1] Strict noninterference ensures that an adversary can deduce nothing about secret inputs from examining public outputs. By definition, noninterference between two processes ensures that no side-channel leaks occur. For example, physically isolating the computing resources of two processes assures digital noninterference. This goal has been difficult to achieve in a practical, efficient manner, but there has been some recent work on the topic.[2]

Given the difficulty of ensuring strict noninterference, there has also been work to relax noninterference with quantitative estimates of how much information actually leaks, enabling leak/risk analysis. This area of study is called *quantitative information flow,* and work in this area typically uses information theoretic measures and simple, mathematically flexible theoretical models.[3,4] Other researchers are investigating extending this theory to complex software systems.[5] Our method is the first practical experimental method to measure information leakage in complex hardware systems. Our methodology also has the advantage that it can be used during early design stage. Since security is a full system property, it is necessary to understand leakage at all levels—from algorithms to high-level language code to system software to architecture and circuit implementation. As such, all the other approaches listed above are complementary to our work.

There has been little work on processor side-channel metrics. The current state of the art is to use qualitative questionnaires for early stage security assessment of processors.[6] To evaluate the security of caches, Domnitser, Abu-Ghazaleh, and Ponomarev have proposed an analytical model to predict the amount of information leakage through cache side channels.[7] Our methodology is more broadly applicable as it can be used to determine leakage in any microarchitectural structure.

References

1. A. Sabelfeld and A.C. Myers, ''Language-Based Information-Flow Security,'' *IEEE J. Selected Areas in Communications,* vol. 21, no. 1, 2003, pp. 5-19.
2. M. Tiwari et al., ''Complete Information Flow Tracking from the Gates Up,'' *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS XIV), ACM, 2009, pp. 109-120.
3. F.X. Standaert, T. Malkin, and M. Yung, ''A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks,'' *Proc. 28th Ann. Int'l Conf. Advances in Cryptology* (EUROCRYPT 09), Springer, 2009, pp. 443-461.
4. S. Micali and L. Reyzin, ''Physically Observable Cryptography,'' *Proc. 1st Conf. Theory of Cryptography* (TCC 04), LNCS 2951, Springer, 2004, pp. 278-296.
5. J. Heusser and P. Malacaria, ''Quantifying Information Leaks in Software,'' *Proc. 26th Ann. Computer Security Applications Conf.* (ACSAC 10), ACM, 2010, pp. 261-269.
6. R. Huang et al., ''Systematic Security Assessment at an Early Processor Design Stage,'' *Proc. 4th Int'l Conf. Trust and Trustworthy Computing* (TRUST 11), Springer, 2011, pp. 154-171.
7. L. Domnitser, N. Abu-Ghazaleh, and D. Ponomarev, ''A Predictive Model for Cache-Based Side Channels in Multicore and Multithreaded Microprocessors,'' *Proc. 5th Int'l Conf. Mathematical Methods, Models and Architectures for Computer Network Security* (MMM-ACNS 10), Springer, 2010, pp. 70-85.

shared system features could also be vulnerable. Could simultaneous multithreading (SMT) expose a pipeline side channel? Could contention at the memory controller be a side channel? Does turning off SMT or partitioning the caches truly plug information leaks? What about a load balancer or cache prefetcher? Currently there is no easy way to answer these questions. With little awareness of the dangers of information leakage at the system level, and no known experimental method to identify or reason about leakage during system design, computer architects may be unintentionally creating leaky systems. As a first step to addressing this problem, in a paper that appeared in ISCA 2012,[6] we described Side-channel Vulnerability Factor (SVF), a metric and methodology to measure information leakage in a processor. SVF enables a quantitative, experimental approach to secure hardware design and uses the simulation-based methodology that is widely used in processor designs. The ''Related Work on Reasoning About Information Leakage'' sidebar discusses other work on addressing this problem.

Side-channel Vulnerability Factor

SVF is a metric and methodology for measuring a side channel's leakiness. It is based on the observation that there are two relevant pieces of information in a side-channel attack: the information an attacker is trying to obtain (sensitive data), and the information an attacker can actually obtain. To measure leakiness, we simply want to compute the correlation between these two pieces of information.

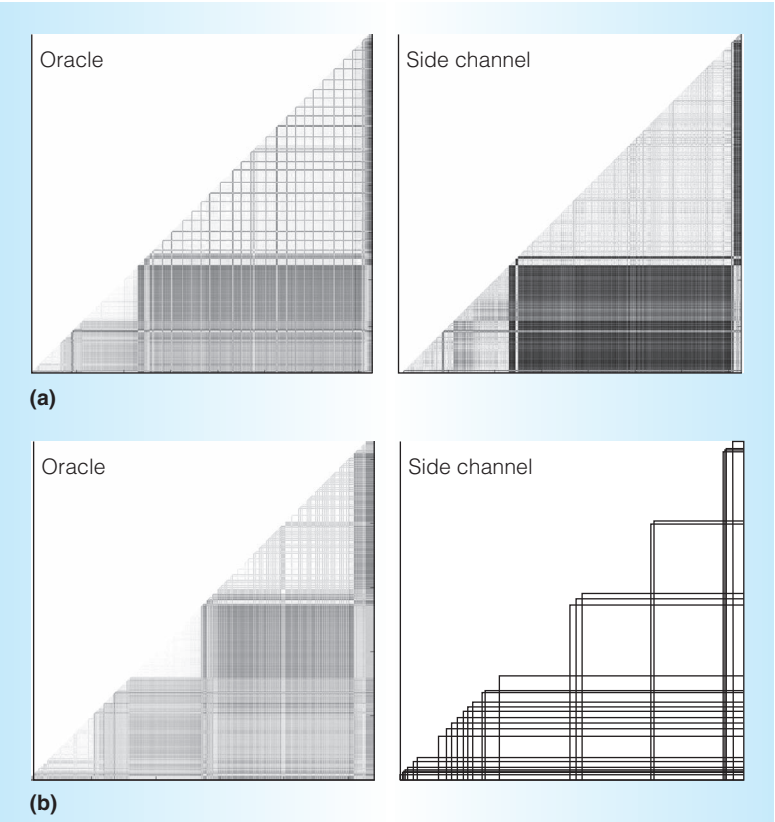


Figure 1. Visualization of execution patterns. The patterns labeled ''Oracle'' show ''ground-truth'' execution patterns of the victim application, and those labeled ''Side channel'' show patterns observed by the attacker for two different microarchitectures. One can visually tell that the high Side-channel Vulnerability Factor system (a) leaks more information than the low SVF system (b).

While measuring correlation sounds easy, it is complicated by the fact that an attacker's observations may not be directly comparable to the secrets. For instance, in one type of cache side-channel exploit, the attacker measures access latencies to each cache line. These measurements effectively probe a victim's usage patterns in various cache sets. However, the secrets with which we want to compute correlation are higher-level data, such as encryption keys, keystrokes, or memory addresses being accessed.

To solve this problem, we recognize a commonality in side-channel attacks: the attacker always uses patterns in the victim's 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, attackers have used memory access patterns in OpenSSL (a commonly used crypto library) to deduce secret encryption keys.[3] These accesses were indirectly observed through a shared cache between the victim and the attacker process. As another example, attackers have compromised crypto keys on smart cards by measuring power-consumption patterns arising from repeating crypto operations.[2]

Patterns have the useful property of being computationally recognizable. Pattern recognition in the form of phase detection[7] is well known and widely used in computer architecture. Intuitively, 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, and computing with a 0 bit is another. By detecting shifts from one phase to the other, an attacker can reconstruct the original key.[1-4] Even nonprocessor side channels such as HTTPS side-channel attacks work similarly—the attacker detects the network phase transitions from ''request'' to ''waiting'' to ''transferring'' and times each phase. In many cases, the timing of each phase is sufficient to identify a surprising amount and variety of information about the request and user session. 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.

Given that we're looking for phase transitions, we use the same type of pattern detection used in the popular SimPoint:[7] comparing each data point to every other data point. This reveals repetitious behavior—phases, in other words. Indeed, these patterns are often visually detectable (see Figure 1a). Furthermore, we want to look at patterns in both the attacker's observations (the side-channel trace) and actual sensitive data (the ''Oracle'' trace). A completely leaky side channel faithfully mirrors the patterns seen in the Oracle trace results. On the other hand, if the original pattern is difficult to discern, as in Figure 1b, it means that the side channel conveys information poorly. As a result, we

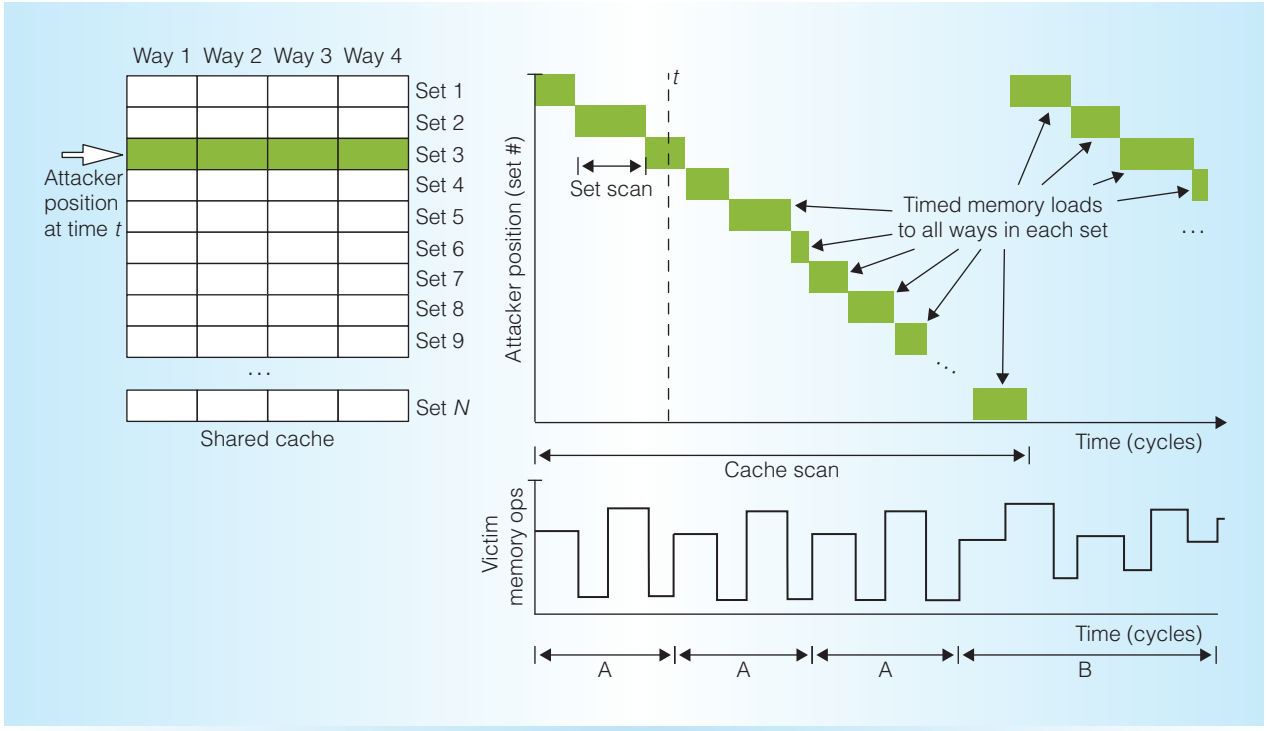


Figure 2. An asynchronous prime and probe attack. An attacker times loads to each set in a shared cache, one set at a time. The amount of time to complete the loads indicates the amount of contention in the set, thus leaking data about a victim's working set.

can determine the presence of pattern leakage by computing a simple Pearson correlation coefficient between the two matrices.

We define the SVF, therefore, as the correlation between patterns in an attacker's observations and patterns in the actual sensitive data which the attack is attempting to obtain. If the attacker's observations have been influenced by the victim's sensitive data, we will observe a nonzero correlation. The greater that influence and the lower the noise in the observations, the higher the correlation. If, however, an attacker's observations contain no trace of the secret data, the correlation will be zero or very close to it.

Case study: memory side channels

Because shared-cache microarchitectures have been exploited in the past,[1-4] it's worthwhile to characterize the vulnerability of cache design space. How do these side-channel attacks work? Are there cache features that obscure side channels? Do protection features reduce vulnerability to undiscovered attacks in addition to known attacks?

Understanding cache attacks

A processor side-channel attack is possible whenever on-chip structures are shared between a victim and an attacker. Figure 2 illustrates a typical cache attack called the "prime and probe" attack, in which the attacker scans the cache during the execution of a victim process. During each scan, the attacker issues loads to each way in a cache set and measures how long it takes to complete all the loads. If the loads complete quickly, it means they hit in the cache from the last cache scan, implying 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. Furthermore, by regularly remeasuring this contention, the attacker measures shifts in the victim's working set. These shifts represent phase shifts in the victim and often implicitly encode sensitive information.

Figure 2 shows an example in which the victim repeats a distinct memory access pattern A. The attacker can't detect the repetition 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 each scan. It is not intuitively clear which factor is more important, though our results imply that speed is the critical factor.

Granularity

In other cases, the victim applications' important phase shifts might occur more slowly than in the example in Figure 2, sometimes much more slowly than the attacker's cache scans. In these cases, an attacker can combine multiple cache scans (by adding them). This combining could help smooth out noise, so information gathered about the victim over larger time spans may be more accurate. 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 that the victim executes during each. Ideally, the intervals align with the natural 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 sizes, beginning with the finest possible: the time it takes for the attacker to complete one scan of the all the cache lines.

Experimental results

We simulated 34,020 possible microarchitectural configurations running the OpenSSL RSA algorithm and measured their SVF and microarchitectural performance. Figure 3 shows a graph of the results. The configurations included different types of attackers, core configurations (SMT on or off and cache sharing levels), cache sizes, line sizes, set associativities, hashing schemes, cache prefetchers, partitioning schemes, and protection mechanisms. (Actual parameters can be found in the original paper.[6]) Additionally, we looked only at relatively fine-granularity SVFs, as they represent leakage of such sensitive information as encryption key bits. We define fine granularity here as less than 10,000 victim instructions, as graphically demonstrated by the gray area in Figure 3.

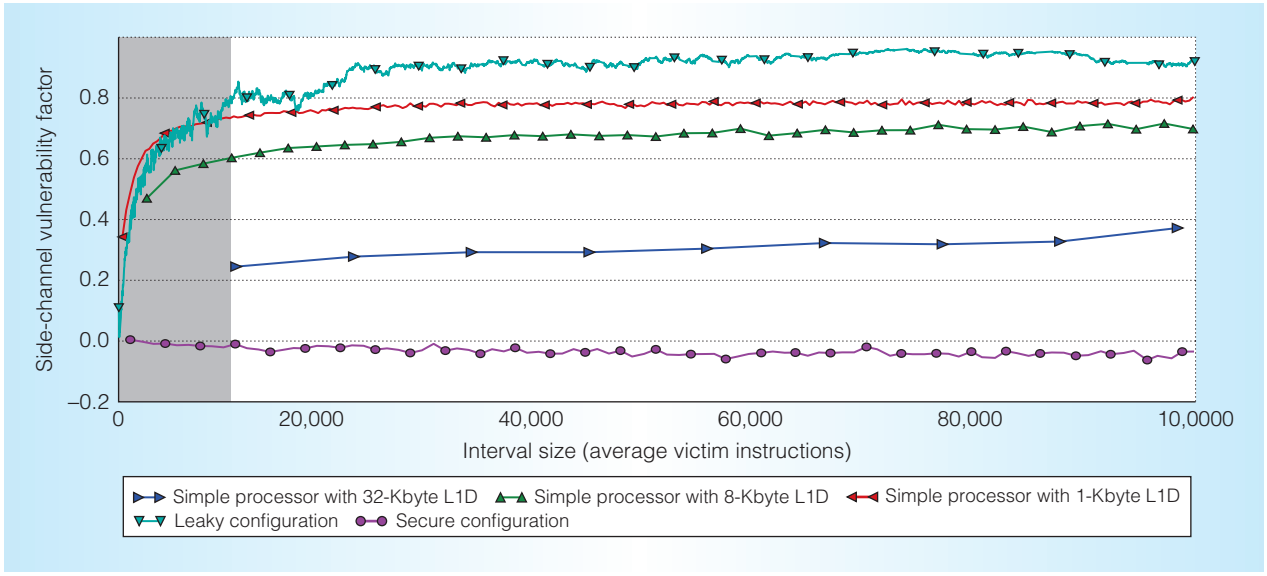
Other interesting results include the following:

- In the 32-Kbyte level-one data (L1D) cache configuration, about 11,000 victim instructions have committed on average in the time it takes for the attacker to scan the cache once, so its line begins at 11,000 on the *x*-axis. As a result, the in-order attacker cannot gather side-channel information leaked during 11,000 instructions. The 8-Kbyte L1D and 1-Kbyte L1D caches, on the other hand, can be scanned much more quickly than the 32-Kbyte L1D cache, and (as a result) much more information is obtained.
- The SVF tends to increase with interval size. This intuitively makes sense; one would expect it to be easier to get accurate information about longer time spans than shorter ones.
- Despite a general trend upward, the 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.

Our results also shed light on some other questions.

Core configuration

Several side-channel attacks exploit SMT capabilities. Accordingly, a first approach to defeating these attacks is to simply turn off SMT to disallow L1 cache sharing. Does this indeed eliminate cache side channels? The histograms in Figure 4 demonstrate that it does not, although it helps. While SMT (which implies a shared L1D) leaks more information to the attacker than no SMT (which realistically means the L1D is



	Configuration		
Component	**Simple**	**"Leaky"**	**"Secure"**
SMT	On	Off	Off
Cache sharing	L1	L1	L2
L1D size	32 Kbytes, 8 Kbytes, or 1 Kbyte	1 Kbyte	1 Kbyte
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 3. The SVFs for several memory-system configurations executing OpenSSL's RSA signing algorithm over a range of attack granularities. The memory subsystem significantly affects the quality of information an attacker can obtain. Note that the configurations labeled ''leaky'' and ''secure'' are not the extremes but simply toward each end of the spectrum. ''Leaky'' represents level-one (L1) cache sharing without simultaneous multithreading (SMT) in the style of core fusion or composable processors.

not shared), there are still many configurations in which an attacker gets information through sharing in the L2.

Cache design-space exploration

Table 1 presents a summary of the results of our design-space exploration. For each possible design choice, the table shows the average, median, standard deviation, minimum, and maximum SVFs of all the designs including that choice. Additionally, it shows the number of configurations in which the attacker can complete cache scans in less than 10,000 victim instructions on average. Only those configurations are included in the aggregates.

The data in Table 1 indicates that some hardware features have a much greater effect on the SVF than others. However, no single design feature makes the system secure or insecure; rather, multiple security policies must be implemented to secure this system.

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

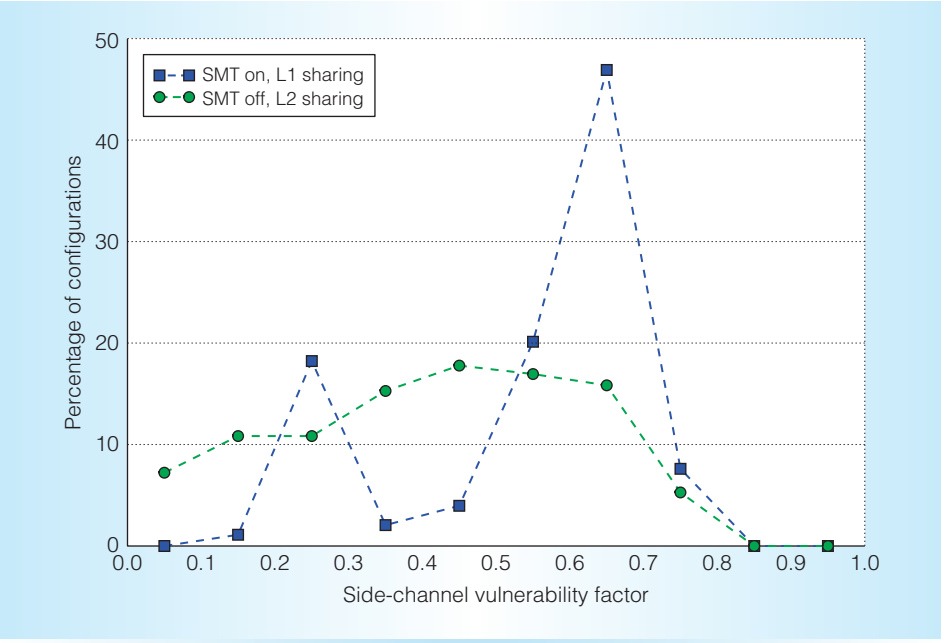


Figure 4. The SVFs of SMT and non-SMT systems. While SMT systems are more likely to be very leaky, non-SMT systems can also leak.

Table 1. Summary of design-space exploration.

Configuration option*		No. of SVF measurements	SVFs				
			Mean	Median	Standard deviation	Minimum	Maximum
All	All	3,933	0.473	0.557	0.222	0.000	0.771
SMT	On	2,493	0.563	0.623	0.158	0.188	0.771
	Off	1,440	0.316	0.328	0.229	0.000	0.735
Cache size	1 Kbyte	2,213	0.499	0.598	0.226	0.002	0.761
	8 Kbytes	1,194	0.475	0.544	0.219	0.000	0.771
	32 Kbytes	526	0.355	0.321	0.163	0.004	0.698
Partitioning	None	1,505	0.511	0.572	0.173	0.031	0.761
	Dynamic	1,170	0.511	0.567	0.168	0.147	0.771
	Static	1,258	0.390	0.432	0.285	0.000	0.744
Hashing	XOR	1,309	0.470	0.569	0.230	0.000	0.771
	Low bits	1,294	0.458	0.546	0.226	0.000	0.744
Line size	8	1,161	0.482	0.600	0.227	0.002	0.736
	64	2,772	0.469	0.554	0.219	0.000	0.771
Associativity	4-way	1,500	0.514	0.606	0.212	0.002	0.761
	Direct	335	0.487	0.567	0.188	0.031	0.689
	8-way	2,098	0.441	0.488	0.228	0.000	0.771
Prefetcher	Next Line	790	0.467	0.519	0.219	0.002	0.756
	Arithmetic	783	0.471	0.561	0.221	0.001	0.765
	None	786	0.487	0.588	0.235	0.000	0.771

*Additional configuration options are discussed in our original paper.[6]

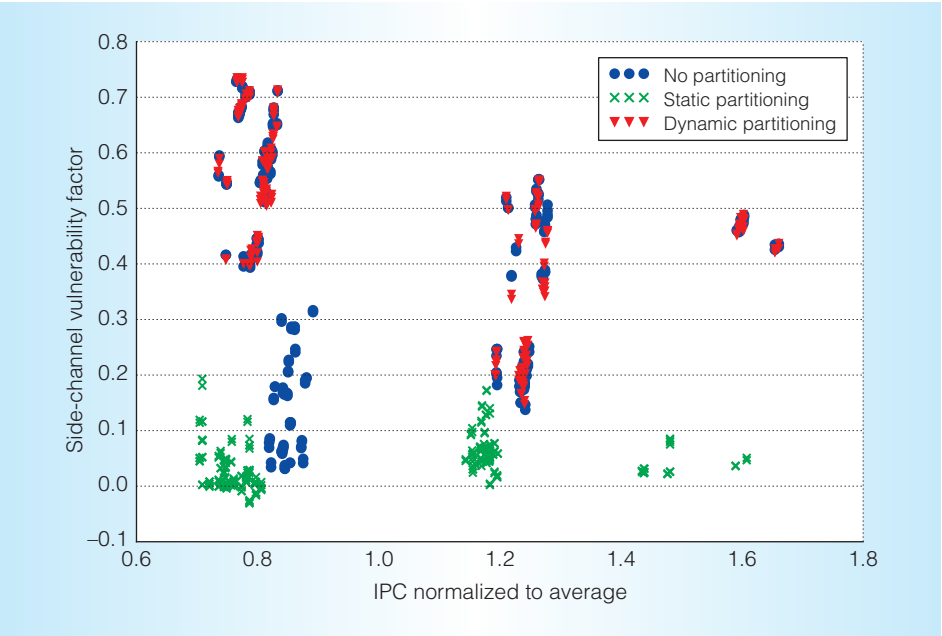


Figure 5. Performance results with various security measures. Many security proposals result in decreased performance, but this need not always be the case. For instance, there are high-performance non-SMT systems with very low SVFs using both no cache partitioning and static partitioning.

their simulation environments to obtain results for their specific designs. Nonetheless, our results suggest a variety of conclusions, which our original paper thoroughly analyzes.[6] Among the important, high-level conclusions, however, are the following:

- Any shared structure can leak information. Even structures intended to protect against side-channel leakage can increase leakage.
- No single cache design choice makes a cache completely secure or vulnerable. Although some choices have larger effects than others, several security-conscious design choices are required to create a secure shared system.
- Cache leakiness is not a linear combination of design choices. Some features leak information in some configurations but protect against it in others. Others offer effective protection only in certain situations. Predicting this leakiness is, therefore, extremely difficult and probably requires simulation and quantitative comparison of the sort we describe.

Some protection features may incur a performance penalty. For instance, random eviction often makes systems more secure but degrades performance. Figure 5 shows performance tradeoffs in non-SMT processors. We actually observe a very weak positive correlation between security and performance. Further, there are configurations with both a low SVF and good performance. As we mentioned earlier, faster victim execution often means lower leakage, as it is harder to observe a moving target. Our conclusion, therefore, is that performance and lower leakage need not always be traded off.

The applicability of the SVF extends beyond caches. To examine other processor structures, a researcher must simply determine what data to use for the attacker's and victim's time series and what distance functions to use to build the similarity matrices. Furthermore, it may be possible to create tools that systemically apply the SVF to many parts of a processor or even larger system and proactively search out leaks. To enable the easy application of the SVF to other systems, we have released a

library for computing the SVF (http://castl.github.com/libsvf). This library can be integrated into architectural simulators and even accommodates measuring non-cache or nonarchitectural side channels. For instance, the SVF could likely model side channels ranging from pipeline to network contention.

The SVF as defined so far, however, is not perfect. As we discuss in our original paper, it has a number of caveats and limitations.[6] For instance, some of the statistical tools are relatively simple, and more complex statistical techniques can likely be applied to solve these problems. It is also not clear how (or if) the SVF could help designers create more secure systems. Since the SVF only provides an entire-system metric, it is difficult for a designer to know exactly which component is leaking and why. Further work is necessary to figure out how to break the SVF down into more useful, finer measurements about individual components. Finally, we have not been able to prove that the SVF is actually correlated to some concrete notion of system security. Unfortunately, this remains an extremely difficult proposition, owing largely to a lack of clear definition of "security" in the context of side channels. While difficult, progress on this particular problem would likely yield breakthrough results.

Regardless of remaining challenges, we find one very important conclusion: side-channel vulnerability is extremely difficult to predict intuitively. Information leakage cannot be completely modeled by linear combinations of leakages in system components or simple theoretical models. Rather, simulation is likely necessary to understand complex system interactions that can lead to leakage. As such, only an end-to-end analysis that accounts for system-level effects—such as the SVF—can accurately determine side-channel vulnerability. MICRO

Acknowledgments

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, Xilinx, and Synopsys. 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. Y. Zhang et al., "Cross-VM Side Channels and Their Use to Extract Private Keys," *Proc. 2012 ACM Conf. Computer and Communications Security* (CCS 12), ACM, 2012, pp. 305-316.
2. S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards,* Springer, 2007.
3. C. Percival, "Cache Missing for Fun and Profit," *Proc. BSDCan 05,* 2005; http://www.daemonology.net/papers/htt.pdf.
4. D.A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," *Proc. Cryptographers' Track at the RSA Conf. Topics in Cryptology* (CT-RSA 06), Springer, 2006.
5. D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games—Bringing Access-Based Cache Attacks on AES To Practice," *Proc. IEEE Symp. Security and Privacy* (SP 11), IEEE, 2011, pp. 490-505.
6. J. Demme et al., "Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage," *Proc. 39th Int'l Symp. Computer Architecture* (ISCA 12), IEEE, 2012, pp. 106-117.
7. T. Sherwood et al., "Discovering and Exploiting Program Phases," *IEEE Micro,* vol. 23, no. 6, 2003, pp. 84-93.

John Demme is a PhD candidate in the Computer Architecture and Security Technologies Lab at Columbia University. His research interests include program characterization and data-intensive computing systems. Demme has an MS in computer science from Columbia University. He is a member of the ACM.

Robert Martin is a software engineer at Google. His research interests include computer architecture security and systems architecture. Martin has an MS in computer science from Columbia University, where he performed the work for this article.

Adam Waksman is a PhD candidate in the Computer Architecture and Security Technologies Lab at Columbia University. His research interests include computer architecture and hardware and computer systems security. Waksman has an MPhil in computer science from Columbia University.

Simha Sethumadhavan is an associate professor of computer science at Columbia University, where he leads the Computer Architecture and Security Technologies Lab. His research interests include hardware support for security and energy-efficient computer architectures. Sethumadhavan has a PhD in computer science from the University of Texas at Austin.

Direct questions and comments about this article to John Demme and Simha Sethumadhavan, 1214 Amsterdam Ave., MC 0401, New York, NY 10027; jdd@cs.columbia.edu, simha@cs.columbia.edu.