Theme Article: Top Picks

Trace Wringing for Program Trace Privacy

Deeksha Dangwal
University of California, Santa Barbara

Weilong Cui
Google, Inc.

Joseph McMahan
University of Washington

Timothy Sherwood
University of California, Santa Barbara

***Abstract*—A quantitative approach to optimizing computer systems requires a good understanding of how applications exercise a machine, and real program traces from production environments lead to the clearest understanding. Unfortunately, even the simplest program traces can leak sensitive details about users, their recent activity, or even details of trade secret algorithms. Given the cleverness of attackers working to undo well-intentioned, but ultimately insufficient, anonymization techniques, many organizations have simply decided to cease making traces available. Trace wringing is a new formulation of the problem of sharing traces where one knows *a priori* how much information the trace is leaking in the worst case. The key idea is to squeeze as much information as possible out of the trace without completely compromising its usefulness for optimization. We demonstrate the utility of a wrung trace through cache simulation and examine the sensitivity of wrung traces to a class of attacks on Advanced Encryption Standard (AES) encryption.**

Privacy in the digital age has become increasingly difficult to achieve and a contentious topic. As technologies that capitalize on facial recognition, location services, and personal health tracking become mainstream, addressing these complex privacy issues is of foremost importance. Policy makers have put in place regulations on data protection through the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). Computer scientists and engineers must develop systems and tools for embedding privacy into existing and new workflows. In this article, we describe a new approach to privacy, wringing, with particular applicability to the problem of sharing program traces.

When working toward application-tuned systems, developers often find themselves caught between the need to share information (so that partners can make intelligent design choices) and the need to hide information (to protect

Digital Object Identifier 10.1109/MM.2020.2986113
Date of publication 8 April 2020; date of current version 22 May 2020.

proprietary methods or sensitive data). One place where this problem comes to a head is in the release of program traces; even the simplest memory access traces leak a tremendous amount of information. For example, we can capture the memory access behavior of a critical cryptographic function (which is known to be a function of the secret key), a set of lookups corresponding to the parsing of a social security number, or even detailed system configuration parameters that are considered a trade secret. While the sharing of these traces between technology partners can lead to more robust and high-performance systems, it can also leak highly sensitive information, and expose user data to security vulnerabilities. Today when such traces are needed, programmers may be asked to obfuscate the key algorithm behaviors to hide sensitive data or provide models of the system, which approximate the same behavior but omit sensitive parts. Hand-built models of the system are both tedious to code and of limited predictive power. Since there is no well-defined and well-trusted approach to this problem, developers are often forced to resort to rough human-language descriptions of the behavior of programs (e.g., it is 80% pointer-chasing). This leads to missed opportunities, frustrated optimization, and the design process ultimately suffers. Ideally, engineers would access methods to eliminate any sensitive information from the traces while still capturing the program behavior and its interaction with the underlying hardware. However, the extent to which "sensitive" data influences program behavior is rarely understood by a single party, and even harder to argue is that it is completely absent from a trace.

We present a new formulation of this problem of sharing traces where before release one knows (*a priori*) exactly how much information a trace is leaking in the worst case. The key idea, wringing, is to squeeze as much information as possible out of the trace without completely compromising its utility. In the ideal case, only the *useful structure* of the trace remains and all potentially sensitive data has been eliminated. While there is no known mechanism of quantifying the amount of sensitive data that remains in an arbitrary trace, we can at least say how much *total* information is shared, which provides a useful upper bound. If we share only a couple thousand bits about a trace, we can then be certain we are not giving away every user's social security number by accident. Reconstructing a useful trace from a few thousand bits of information is hard, but interestingly we are free to use any *public* information about the nature of these traces in helping us accomplish this. Compression, when taken to this extreme and lossy form, connects to privacy in this unexpected way. However, as is often the case in computer architecture, an important tradeoff remains between information leaked and ability of the trace to capture the program behavior.

The key idea, wringing, is to squeeze as much information as possible out of the trace without completely compromising its utility. In the ideal case, only the useful structure of the trace remains and all potentially sensitive data has been eliminated.

We formalize this new approach specifically in the context of memory address traces in part because we have many prior trace analysis techniques to build on.[7,9,12] To expose the tradeoff inherent to this problem, we explore a new class of memory trace synthesis techniques based on ideas from signal processing. By projecting the address space onto a wrapped 2-D heatmap, we decompose memory behavior into orthogonal set of features that can then be replayed to reproduce the same "visible" patterns as the traces under examination. Specifically, we use a Hough-transformed[3] trace to find both constant and strided access patterns. We find that for memory traces it is indeed possible for useful program behavior to be conveyed in only a few thousand bits. We demonstrate the utility of wrung traces through cache simulation with bounded leakage, and even examine the sensitivity of wrung traces to a class of attacks on AES encryption.

TRACE WRINGING AS A NEW GAME

The program traces we look at in this article are memory access traces specifically, but more generally fall into a class of traces useful for application-tuning and hardware–software

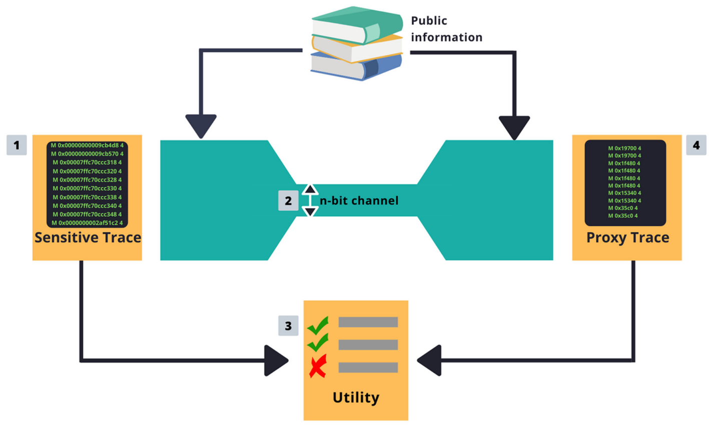

Figure 1. Forcing a trace through a channel with a capacity of only a few bits bounds the amount of sensitive data shared. While public information such as prior non-private traces can be used in the creation of the code, the trace to be coded must not be known to the receiver. The objective is to minimize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

co-optimization (as opposed to debug traces). A program trace can contain a tremendous amount of information about the system under evaluation. But, as we know, such traces are invaluable for performance evaluation because they demonstrate the way the system actually behaves in the face of the workloads it must actually handle. While the behaviors are important at a high level, rarely are the specific elements of the trace critical. Rather it is the relationship between those elements and the proportions that they appear in the trace that is often the key. This is of course not a new insight; what we claim as new is the idea that we can formalize these schemes in such a way that it bounds the amount of information leaked about a system being traced.

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

Answering this question requires an analysis across two metrics: information leaked and utility, as described in Figure 1. Information is surprisingly easy to quantify; it is the number of bits from the secret trace that needs to be "transmitted" between the full trace (which contains every address) and proxy trace (which is a stand-in for the full trace and is ready for release). In Figure 1, Step 1 is to *encode* the secret trace. Note that any information from public traces or training data can be shared freely and even hard-coded into the "receiver," but in the end, everything you wish to share about the full trace must be represented in a single n-bit "packet" (Step 2 in Figure 1). Quantifying utility is harder and more use-case specific. For memory address traces, we define a distance function between cache miss-rates of trace vectors as one such function (Step 3 in Figure 1), but, in general, there are many other metrics one might use.

SIGNAL PROCESSING APPROACH TO WRINGING

Given the above mentioned constraints, the question is how to encode memory address trace behavior in a general, and yet incredibly compact, manner. Our compact representation must also capture the *structure* of these traces so that we can identify, describe, and quantify the patterns that we care most about. We present a signal processing pipeline for trace wringing. Our approach describes traces as a probabilistic grammar of generators coupled with very high level accounting of behavior over time. The "transmitted" bits encode both the structure and parameters of this scheme.

To understand our approach at a high level it is useful to start with a visual sense for the structure of such traces. We project the address trace onto a fixed-size modulo-mapping of the memory spaces to create a *heatmap*. Figure 2 shows such a heatmap for gcc where instruction count (time) runs along the *x*-axis and the address runs along the *y*-axis. If we were to plot this for the *entire* memory, it would clearly be too large for such a graph (the distance between the stack and heap would dwarf any local behavior), so instead, we plot the address modulo a large power of two. Heatmaps such as this have the advantage of mapping addresses onto a more manageable space, but at the same time, keep the spatio–temporal structures that would actually impact a real cache.

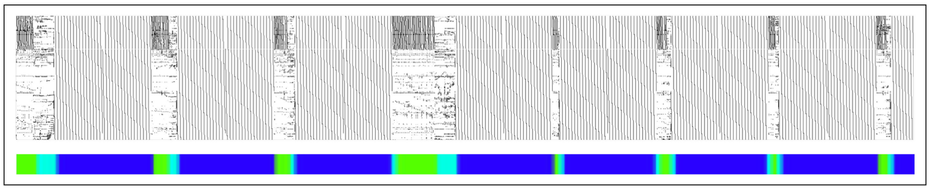
Figure 2. Phases visible in the trace generated by gcc after k-means clustering. Each of the three colors in the bar marks a unique phase in the trace. Note, importantly, that phases reoccur over time.

Interesting and intuitive patterns emerge after looking over this graph. The flat horizontal lines in the graph are patterns of repeating access to a set of addresses. These are high temporal locality behaviors. Sharp diagonal lines, on the other hand, are regions of high spatial locality as addresses are accessed one after the other in succession. If we can concisely capture the *character* of these behaviors, without transmitting the addresses themselves, we can minimize the amount of information leaked. The modulo-memory heatmaps exhibit hierarchical organization. Globally, there exists a recurrence of similar patterns in the order of a few tens of thousand instructions, i.e., the presence of program phases, and within them, we observe patterns that we associate with the more local memory access activity. In order to find some representative of the higher echelons of this hierarchy, we employ k-means clustering for program phase analysis.[2,9] Rather than encoding the entire trace monolithically, we can encode just the k representative clusters independently. By breaking the pattern down into a set of simpler behaviors, we can then tackle them one-by-one. Figure 2 shows the result of running the phase detector on the memory address trace for gcc. Each of the three colors in the bar in the figure show the occurrence of three unique phases in the memory access trace. The technique does a good job of lining up with the repeating structures in the heatmap. With these phases marked, we can encode the k representative clusters with $\log_2 k$ bits.

Given that both strong temporal and spatial locality features show up as lines, decomposition into a set of line segments is a natural place to start. The Hough transform can be used to then find the locations and orientations of certain geometric primitives, such as lines, in the given space. We apply Hough transformation, a popular computer vision technique for detecting patterns in images; for our features, we employ the Hough line transform. Specifically, we use the progressive probabilistic Hough transform,[5] a rendition of the Hough transform algorithm that only performs voting on a subset of the input points. These input points are chosen based on certain features of the expected result, such as a threshold, the length of the expected line, interpolation strategies, and the angle of the line. By interleaving the voting process with line detection, this algorithm finds the most prevalent features first, while also minimizing the computational load. The progressive probabilistic Hough transform returns a set of lines, with each lines (*x*,*y*) coordinates in the modulo-memory heatmap space. We also introduce a variable, weight, for each line, which is a measure of darkness of the line. Some intuition about how the probabilistic Hough transform functions is described in Figure 3.

The list of phase identifiers (the result of clustering), the two (x, y) coordinates of each line segment detected by the Hough transforms, and the line's weight in the representative phase, create compact "information packets." The size of the total "transmission" is n and bounds the maximum amount of information leaked.

After phase detection and Hough-line transformation, we end up with a set of lines for each representative phase. We can see the decomposition of a phase of gcc into lines in Figure 4. Each phase

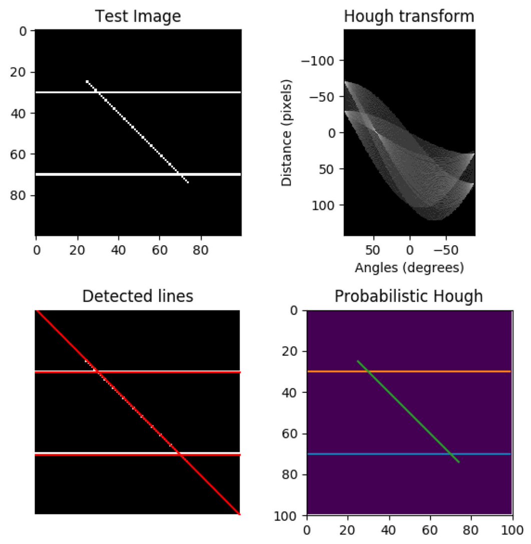

Figure 3. We capture information about lines we observe in trace heatmaps using the Hough transform. Here, we demonstrate its working. The points on the test image are surveyed for parameters in the polar coordinate space described as the Hough transform. The intersections describe the parameters of the detected lines. The final figure shows the probabilistic Hough lines, the more robust and efficient algorithm. For our heatmaps, we use the probabilistic Hough line algorithm.

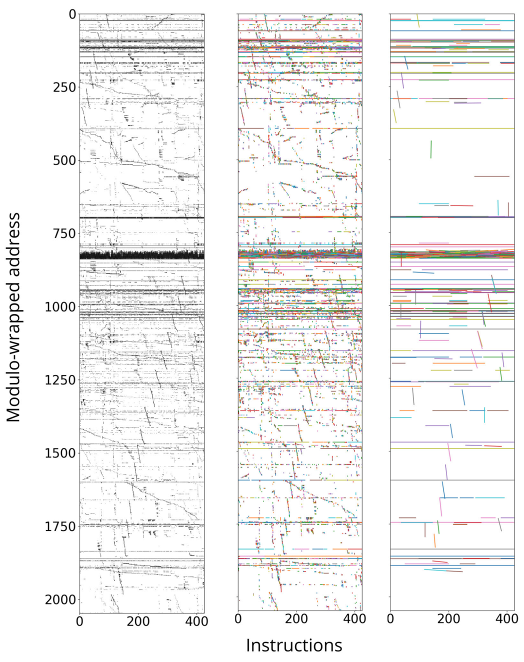

Figure 4. Producing probabilistic Hough lines[5] on top of the heatmap for a program phase in gcc. The colors are used to indicate distinct lines produced by the decomposition. Line length, threshold, line gap, and line slope are input parameters that can be used to adjust the tradeoff space between information leakage and utility. For example, choosing a small line length will ensure that even the smallest lines are captured, but at the cost of transmitting information about even the smallest features. Here, we see how the choice of parameters can be used as a knob to control the features we want to encode to find a suitable tradeoff point.

is also assigned a label indicating to which cluster it belongs to, i.e., which representative phase "represents" it. Since the structural information of each phase is encoded in the Hough lines, we can generate an "address tracelet" for each phase using the representative's lines. Phases from the same cluster may occur intermittently and in different lengths. For all phases in the same cluster, we generate patterns continuously in a rotating fashion regardless of the length. Upon picking a Hough line at time t, we generate an address "segment" from that line based on a fixed segment length, which captures locality at a small granularity. The final proxy trace has the same length as the original trace and captures the most salient aspects of its behavior while at the same time leaking no more than n bits.

EVALUATION AND OVERVIEW OF RESULTS

In our pipeline, we pose the problem of sharing traces between technology partners with two subsystems. The trace-wringing subsystem minimizes the number of bits used to describe the trace structure, and the generator subsystem uses budgeted information to generate the proxy trace. Trace wringing includes the generation of heatmaps from memory access traces, phase analysis, decomposition of representative phases into lines, and creation of packets to transmit this information. Note that there are actually many possible values of n and different parameters will result in different tradeoffs between utility and privacy.

To evaluate the effectiveness of the approach, we take a subset of the SPEC2006[6] traces, wring them through our pipeline to a target number of bits, a *bit budget*, and evaluate the traces across a range of cache

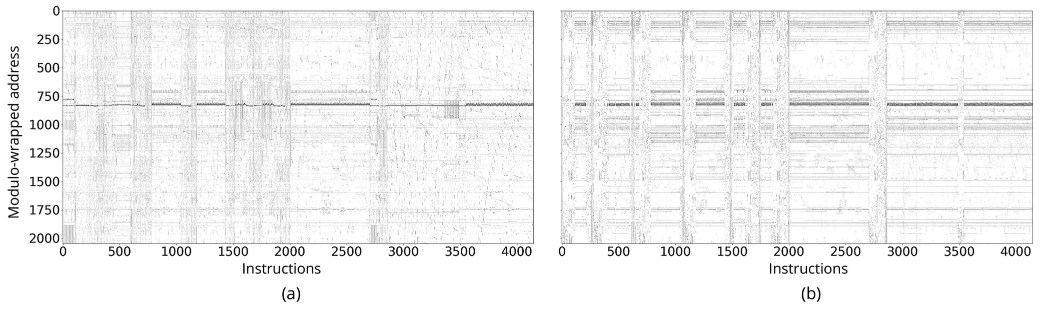

Figure 5. (a) Heatmaps for the sensitive input trace gcc and (b) the trace-wrung proxy generated by our pipeline. The heatmap of the trace-wrung proxy shows that both global and local features line up with the input trace. All but the subtlest of patterns are present in the trace-wrung proxy.

configurations with regards to miss rate. We indeed observe that as the bits of information leakage increase, the proxy miss rate gets closer to the ground truth miss rate, which confirms that with more information shared, the proxy trace we reconstruct becomes more similar to the original trace in terms of structure. The exact nature of the tradeoff space is explained more in the paper. In our setting, we define utility in terms of similarity of cache miss rates. We measure and present cache miss rates in the article. In Figure 5, we compare the proxy heatmap generated for gcc, against the ground truth. Our wrapped address space is of height 2048 (cache lines in the heatmap) and each column in the heatmap corresponds to 10 000 memory accesses. The figure illustrates that our approach is able to capture all but the subtlest of patterns.

It is also worth noting that we further explored real attacks on the resulting traces. Specifically, we choose to examine the trace to see if it is possible to recover an AES key using known attacks. AES attacks based on cache sets have been well-studied.[8] We present the details of the attack in our paper. We find that trace wrung proxies completely stop traditional AES attacks. We perform this attack on a set of traces collected from runs of AES with a random plaintext. We perform the same attack pre- and post-wringing. Prewringing, the attacker correctly guesses the upper 5 bits of all 16 key-bytes after 1838 encryptions. This is the maximal information that can be learned in a first-round attack with 8-byte cache lines. Postwringing, the attack guesses wrong for all 16 bytes of the key after 50 000 traces.

CONCLUSION AND FUTURE IMPACT

Looking further out, the conflict between the need to share information (to provide more optimal performance) and hide information (for privacy) is becoming increasingly fundamental in all of computer science. Threats to personal data privacy are emerging as a leading concern for users. While the European GDPR and CCPA put in place privacy and data protection requirements, the onus of implementing tools to understand and embed privacy into systems falls on engineers. Computer architects must start thinking more about privacy and provide infrastructure to enable privacy at all levels of computing. Trace wringing can be leveraged as one such tool.

We feel the tradeoff space exposed by trace wringing will open the doors to future work at the intersection of privacy and computer systems optimization. We hope to see more follow-up work that builds on years of community experience dealing with address traces and to encode common patterns in a general way. In many applications, striding memory behavior is an important component and we believe we are the first to connect the address trace analysis problem with the Hough transform. The resulting analysis is surprisingly robust to noise and can capture general striding behavior. While this approach is effective for the memory problems we examined, there is no shortage of opportunity to build on the techniques we lay out to create more robust and higher quality trace wringing systems. Fully leveraging the best synthetic trace generation,[11] trace compression,[1,4] quantitative information flow,[10] and statistical modeling[12] techniques and understanding what they each bring to the problem is one next step. Bringing the full algorithmic power provided by the fact that *any* public trace data can be leveraged in the compression is also very promising. This opportunity is particularly interesting as it sits outside of any past lossy compression or synthetic trace scheme's ability to exploit (i.e., minimizing total data transferred is different than minimizing sensitive data transferred).

We feel the tradeoff space exposed by trace wringing will open the doors to future work at the intersection of privacy and computer systems optimization. We hope to see more follow-up work that builds on years of community experience dealing with address traces and to encode common patterns in a general way.

In application-tuning and system design, one can certainly understand how related problems exist with storage traces, cache coherence traffic, energy usage, user interaction data, and certainly location data. Clever, yet complex, techniques have been developed to address certain anonymity problems in the past, yet the reality is that they are often dependent on specific assumptions such as a lack of prior information, statistical distributions governing the data, or that number of queries can be tightly bounded. Our wringing approach is very direct and that comes with clarity as to what it does and does not do. It does not *guarantee* anything about how useful the resulting trace will really be for optimization. However, it *does* transform the problem of safe sharing into a *measurable* systems problem subject to the myriad tools we have at our disposal for *common-case optimization*. Furthermore, it *does* provide a *strong and clear bound* on the amount of useful information given by the trace. For the purposes of privacy engineering, this is exceedingly valuable. There is a consensus on the importance of building privacy into systems that deal with information about health, legal records and law enforcement, transportation and location, and other sensitive information. But, the capability of today's tools and methodologies is limited. Trace wringing provides evidence that new methods that bound information sharing in useful ways are possible, and perhaps more importantly, they can then be improved when fed back as a system requirement.

REFERENCES

1. M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam, "The VPC trace-compression algorithms," *IEEE Trans. Comput.*, vol. 54, no. 11, pp. 1329–1344, Nov. 2005.
2. A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 217–227.
3. R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Commun. ACM*, vol. 15, no. 1, pp. 11–15, 1972.
4. E. N. Elnozahy, "Address trace compression through loop detection and reduction," in *Proc. ACM SIGMETRICS Perform. Eval. Rev.*, 1999, vol. 27, pp. 214–215.
5. C. Galamhos, J. Matas, and J. Kittler, "Progressive probabilistic Hough transform for line detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 1999, vol. 1, pp. 554–560.
6. J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.
7. P. Michaud, "Online compression of cache-filtered address traces," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 185–194.
8. D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers Track RSA Conf.*, Springer, 2006, pp. 1–20.
9. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 45–57, 2002.
10. G. Smith, "On the foundations of quantitative information flow," in *Proc. Int. Conf. Foundations Softw. Sci. Comput. Struct.*, Springer, 2009, pp. 288–302.
11. L. Van Ertvelde and L. Eeckhout, "Dispersing proprietary applications as benchmarks through code mutation," *ACM SIGARCH Comput. Archit. News*, ACM, vol. 36, pp. 201–210, 2008.
12. J. Weinberg and A. E. Snavely, "Accurate memory signatures and synthetic address traces for HPC applications," in *Proc. 22nd Annu. Int. Conf. Supercomput.*, ACM, 2008, pp. 36–45.

Deeksha Dangwal is currently working toward the Ph.D. degree in computer architecture with the Department of Computer Science, University of California, Santa Barbara. Her research interests include computer architecture, privacy, and information theory. She is a student member of IEEE and ACM. She is the corresponding author of this article. Contact her at deeksha@cs.ucsb.edu.

Weilong Cui is currently a Software Engineer with Google. His research interests include statistical/economic-inspired methods and programming languages for computer architecture performance modeling, as well as novel micro/system-architecture and its interaction with software. Cui received the master's and bachelor's degrees in computer science from Peking University and the Ph.D. degree from the Department of Computer Science, University of California, Santa Barbara. Contact him at cuiwl@google.com.

Joseph McMahan is currently a Research Scientist with the University of Washington, Seattle. His research interests include computer architecture, security, formal methods, and machine learning. McMahan received the Ph.D. degree from the University of California Santa Barbara. He is a member of ACM and IEEE. Contact him at jmcmahan@cs.washington.edu.

Timothy Sherwood is currently a Professor of computer science and the Associate Vice-Chancellor for Research with the University of California, Santa Barbara. He is a cofounder of the hardware security startup Tortuga Logic and the 2016 ACM SIGARCH Maurice Wilkes Awardee "for contributions to novel program analysis advancing architectural modeling and security." Sherwood received the B.S. degree in computer science from UC Davis, and the M.S. and Ph.D. degrees from UC San Diego. Contact him at sherwood@cs.ucsb.edu.