Overcoming the Intuition Wall:

Measurement and Analysis in Computer Architecture

John Demme

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2014

ABSTRACT

Overcoming the Intuition Wall

John Demme

These are exciting times for computer architecture research. Today there is significant demand to improve the performance and energy-efficiency of emerging, transformative applications which are being hammered out by the hundreds for new compute platforms and usage models. This booming growth of applications and the variety of programming languages used to create them is challenging our ability as architects to rapidly and rigorously characterize these applications. Concurrently, hardware has become more complex with the emergence of accelerators, multicore systems, and heterogeneity caused by further divergence between processor market segments. No one architect can now understand all the complexities of many systems and reason about the full impact of changes or new applications.

Instead, computer architects must often rely on approximations of software behavior and hardware operation. By using their intuition when necessary and quantitative methods when possible and feasible, architects can actually function. Historically, however, advancement has been achieved most rapidly through deep quantitative understanding and evaluation of ideas and systems. Indeed, computer architecture has a long history as a quantitative field and has benefited greatly from the use of quantitative methods. Despite a large amount of literature in the area many questions remain unanswered, motivating further research.

To that end, this dissertation presents four case studies in quantitative methods. Each case study attacks a different application and proposes a new measurement or analytical technique. In each case study we find at least one surprising or unintuitive result which would likely not have been found without the application of our method.

In our first study, we present a new technique for using performance counters which

reduces the overhead of counter reads by 23x. This reduced overhead allows us to measure the detailed behavior of several important web applications. The data we present led to a series of recommendations for future systems, many of which were surprising. For instance, modern web applications differ from popular benchmarks in at least several ways, motivating a new set of benchmarks. Although many case studies of application behavior already exist in the literature, our new technique allowed us to examine detailed behavior of production applications with unscaled inputs.

The second case study examines an emerging problem in security called side-channels. In short, the sharing of resources like caches create a channel by which attackers can gain small but crucial pieces of information about other applications; in the most famous cases, attackers can deduce bits in secret encryption keys. One of our case studies attempts to quantify side-channel information leakage, allowing us to compare different systems' security and begin to understand the reasons for this leakage. Our technique can be applied at or before design time, allowing leaks to be caught and repaired before the product goes to market. We have also found a series of surprising results, indicating that intuitive understanding of existing attacks are somewhat superficial. We conclude that quantitative methods like ours are necessary for the study of side-channel information leaks.

In order to deal with the large and ever changing landscape of applications, it is helpful to have methods to examine, understand, and present interesting code patterns to us. Our third case study presents a technique for mining large amounts of code to find common patterns. We present a novel method of approximate graph clustering, which enables the mining of program graphs from large code bases. It also gives us an unbiased way to find similar code across many code bases, beginning to answer questions about how to design accelerators. Short of that goal, it helps architects inform their intuitive understanding of software.

Our final case study investigates the near-universal problem of malware. Despite decades of research and a multitude of commercial products, viruses and their brethren exist and indeed multiply each year. Instead of further traditional detection techniques, we ask if hardware can learn to detect malware based on its behavior. By applying machine learning techniques to data on architectural behavior, we are able to build robust, secure malware detectors. Our approach avoids large amounts of manual work which can lead to buggy code. It also led to very simple detectors which could be built into secure hardware.

The case studies presented here demonstrate the utility of quantitative methods. They further our understanding of systems; allow the rapid, detailed study of new applications; help create systems which are more robust; and guide designers in the creation of hardware. In short, quantitative methods help scale the intuition wall.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Much as I owe thanks to many people for their help and support in my work, I am somewhat uncomfortable writing this section. This dissertation is about quantitative methods – elevating our intuition beyond the qualitative. The acknowledgments, however, are intended to be qualitative, biased, and humanizing. Thus this section is somewhat antithetical. Nonetheless, I owe my colleagues, friends, and family much gratitude, so please excuse the following gushing vulgarity of subjectivity, partiality, and emotionality.

I must first thank my advisor, Dr. Simha Sethumadhavan. There are endless things for which I am grateful – many of which are not atypical, so I will spare the boilerplate. There is one thing he taught me, however, that has shaped me significantly. It is exemplified best by his most common feedback on my drafts, "This needs more structure." Since my arrival as an incoming student, I have come to appreciate structure in text, argument, and thought as a tool for clarity. This philosophy and penchant for organization has and continues to help me clarify my ideas, my positions, and my expression of both. It has given me better intellectual clarity, a gift of immeasurable value.

My colleagues at Columbia have also been instrumental. First, thank you to all of my coauthors – I could not have done it all without you all. Thank you also to Dr. Martha Kim for all of your invaluable feedback on my presentations, paper drafts, and my rebuttals. Thank you to Dr. Steven Nowick for your insightful feedback and your time discussing my projects. Thank you to all of my committee members for your feedback on this dissertation, especially Dr. Alfred Aho for your thorough analysis of my work and alternative perspectives. Thanks to all the members of CASTL and CSL for attending my talks over the years, reviewing my drafts, and your feedback on all my work. Thanks to all of the administrative and technical staff at Columbia, especially to the people in CRF who were often left to clean up after my exploits.

Thanks to all my friends at Columbia. Without our happy hours to blow off steam, I would not have made it through – the trade of brain cells lost to alcohol in exchange for continued sanity turned out to be a necessary Faustian bargain. Thanks for our discussions about ideas, directions, and all the potpourri. I also want to thank all of my friends outside of Columbia, both those I have known most of my life and those I have known for only the last few years. Your continual support and friendship has been indispensable.

Perhaps the bulk of my gratitude goes to my parents and family. There are no words. My parents have continually supported me for as long as I can recall – morally, emotionally, financially, and in every other way imaginable. All of my achievements are in part their achievements.

Finally, to anyone amused by the opening to this section: thanks for being one of the few to laugh at my horrible, horrible jokes over the years. I have interacted with countless people over the course of my academic career and indeed my life, many of whom have influenced me in some respect. I can only hope to have paid back my share on average and will continue to do so in the future.

Chapter 1

Introduction

The landscape of computing has changed radically over the last several decades. While traditional applications of high performance computing and business intelligence still exist, computing has become far more pervasive. It now alters social interactions; acts as a constant aide in our pockets; provides new venues for criminal activity; deeply embeds itself to provide reams of data about nearly everything; learns and teaches us new things about both ourselves and our world. The list of new applications and the raw variety of computational tasks could stretch from here to infinity. In response, computing systems have increased in complexity, especially in the mobile segment where systems on chip can be highly heterogeneous. Even without processor heterogeneity, multi-cores and virtualization in data centers create highly heterogeneous workloads, leading to complex interactions at the hardware level.

As a result of the plethora of applications and matching complexity in hardware, it is no longer possible for a single software engineer to reason about an entire system with any level of detail. One struggles to understand the operation and hardware interaction of a single program, let alone develop intuition about the operation of many. Similarly, hardware engineers cannot possibly understand and deduce the detailed operation of an entire hardware system – they simply have too many components. For most, however, this inability to internalize the operation of the entire computing universe is not important. A software engineer need worry only about the correct and efficient operation of his software. A hardware engineer typically need only design her component, working to its specification.

On the other hand, for computer architects heterogeneity in applications and complexity in hardware poses a major challenge. How can one design an efficient, performant computer without thoroughly understanding all of the applications it might run? How does one understand and predict the detailed operation of entire computing system without knowing the detailed operation of all its parts? Of course, one simply cannot and does not thoroughly understand all aspects of a system. Instead, one operates using approximations of application and system behavior. We postulate that a result of this approximation is the intuition wall – that the quality of our systems' designs are in part limited by the amount of detail about applications and hardware which we can intuitively understand. The more accurate our intuitive approximations and the more we understand about the software and hardware systems, the better our systems will be.

Thesis & Contributions To scale the intuition wall, we suggest that the architecture community must further our quantitative understanding of systems with more and better techniques for measuring and analyzing systems. This is not a new idea. For decades, computer architecture has been a deeply quantitative area. There is a vast quantity of research in program analysis and microarchitectural performance evaluation. While continuing to innovate on measurement and analysis for traditional architecture problems, we must also extend our work into the new domains which architecture is beginning to examine. To this end, this dissertation presents work on techniques for measurement and analysis in both a traditional domain (program analysis) and a new one, hardware-based system security.

1.1 Does the intuition wall exist?

Clearly, no intuition about what software does or how hardware works is perfect or all-encompassing. But does lack of intuition or inaccurate intuition hinder innovation in computer architecture? Would superior characterization methodologies lead to faster development and faster chips? Would replacing manual design and/or design evaluation with automated techniques help? Almost certainly, yes. The more interesting question is to what degree? We argue that many important issues in computer architecture rest on the accuracy and breadth of information about hardware and program behavior.

Benchmarks Instead of laboriously examining huge numbers of applications, architects typically rely on benchmark application sets like SPEC [67] or PARSEC [18]. While this makes architects' jobs feasible, it requires us to assume the benchmarks to be representative of all other or a class of applications. Clearly, this assumption is wrong for some values of "representative", but may be reasonably close to true for others. For example at an extreme: designing highly specialized, fixed function units for benchmarks would not benefit real-world applications. Benchmarks are reasonably representative, however, in that adding general purpose features like branch prediction may result in roughly similar speedups for both benchmarks and real applications. As a result of these extremes, the representativeness of benchmarks is often treated skeptically [136]. In fact, it has been quantitatively shown that the popular SPEC benchmark suite is redundant [129] and indeed neither SPEC nor PARSEC are representative of some important workloads [55]. In the latter case, more efficient architectures were designed based on those applications' differences from standard benchmarks [106]. One could conclude that reliance on benchmarks – approximations of real-world program behavior – limits the utility of innovations.

Quickly Understanding New Applications There also exist many questions for which answers can be found, but require large amounts of manual effort. For instance, understanding what code does and how it operates is extremely laborious. In forensic analysis of malware, for example, well trained people must dissect binaries and examine their execution. Another example: architects often must run very slow simulators to gain detailed information about the performance characteristics of programs. In both cases, tools to assist or speed up the tasks would drastically decrease the effort in understanding important phenomena and thus increase the rate of innovation and/or the breadth of applications which can be studied.

Evaluation of Results Most of computer architecture is deeply quantitative; we evaluate designs and ideas with objective metrics like wall clock time and energy consumption. In new areas, however, metrics have yet to be defined because sometimes it is unclear how to evaluate an idea. One area in particular – security – often cannot be quantitatively evaluated. Indeed, how *does* one measure the security of a system? No obvious, objective metrics exist. Instead, many security proposals are judged by two measures: (1) do they defeat existing attacks and (2) is there intuitive reason to believe they would defeat future attacks? In many cases, it is trivial to defeat existing attacks while not necessarily increasing security, requiring us to judge security techniques based on intuition alone. As a result, many proposed solutions to security problems are broken within months of publication. Without objective metrics by which to judge proposals it is difficult to make true progress.

Accelerators One growing field of research is hardware accelerators. Two of the most critical and interesting questions in accelerators are what should be accelerated and how programmable should the resulting device be? When one accelerates only a very specific algorithm due to overwhelming use (like a video decoder), these questions are relatively easy to answer. Should one wish to target a class of algorithms or applications with a single accelerator, the problem becomes much harder. What range applications can be sped up by common hardware? How common are those applications? Do there exist common patterns in applications that can be targeted? *None of these questions have been sufficiently answered – intuitively or quantitatively.* However, determining these answers is clearly a roadblock to determining the range and importance of acceleration.

1.2 Motivations

How important is the intuition wall? Computing is a thriving industry, built using the quantitative methods and intuition already in place; why need we develop either further? We argue that two new trends – specialization and worsening silicon economics – make the intuition wall, and computer architecture in general, more important today than in the past.

1.2.1 Specialization

Traditionally, microarchitectural speedups have benefited nearly all applications; however, new proposals and directions in computer architecture have smaller ranges of applicability. Caches, for instance, have some benefit even for applications which exhibit only very little locality. Likewise, one is hard pressed to find a program which is not sped up by branch prediction. Today, however, adding general purpose microarchitectural resources to speed up serial execution – Pollack's rule scaling – doesn't always make much sense in an era with significantly constrained power budgets [50, 69, 159], so other approaches (like multicores and accelerators) have been adopted. Unlike many previous innovations, new hardware designs don't float all boats; some applications benefit, while others can experience significant slowdown.

Multicore parallelism, for instance, can only be applied to certain algorithms. Further, when adding more cores to a die while maintaining constant area and power, serial performance drops, slowing down applications which are not or cannot be parallelized. Even how to best apply parallelism varies wildly from application to application and so compiler, library, and architecture performance vary greatly with application needs.

There may not exist a small set of applications which truly represent most others. Indeed, even benchmark sets are specialized: SPEC [67] for serial applications, Parsec [18] for recognition mining synthesis (RMS) parallel applications, CloudSuite [55] for modern server workloads, et cetera. Instead of treating a small suite of applications as representative, we must often become domain experts in only one or a few types of algorithms. In a time when architectural choices can have such a drastically application-dependent effect on performance, it seems important to be able to accurately understand the operation of a large breadth of applications.

1.2.2 Economics

Modern architects deal with a variety of problems arising from poor technology scaling – the memory wall, the power wall, slowing clock rates, and the potential end of Moore's law. Few of these problems are truly new, however. BJT technology gave way to PMOS logic due to superior power and scaling properties. PMOS eventually lost to NMOS and NMOS to CMOS for similar reasons. CMOS in particular became popular (and has remained so) due largely to its far lower power consumption. Aside from power and scaling, computer engineers have always grappled with order-of-magnitude differences in access time differences between different levels in the memory hierarchy. Actually, were memory size held constant, memory latency has held essentially constant over the decades – an L1 cache of several kilobytes can be accessed in several cycles, just as a similarly sized main memory from the 1980s. Are there any truly new pressures on computer architecture?

One area of chip design which is often overlooked, drives virtually all industry, and is trending in negative directions is economics. Semiconductor manufacturers are famously secretive about pricing, so quantitatively evaluating semiconductor economics is virtually impossible. However, it is well known that the amount of money required to develop each new technology process tends to increase. Further, according to NVidia, future process technologies will not decrease the price per transistor as they usually do [74]. Further, the design costs and complexity associated with each chip design increase with technology scaling: design rules tend to get more complicated, more transistors must be laid out, clock trees get larger, power budgets more difficult to manage, et cetera. In current technologies, anecdotal estimates of design costs start at 10's of millions of dollars, and increase dramatically from there. The cost of a single set of masks – the key physical device which defines a design – could alone cost as several million dollars, though accurate figures are not publicly known. As a result, the cost of merely attempting to innovate is extremely high, so companies must be absolutely convinced of an innovation's benefit before building it.

Concurrent to worsening economics, Moore's law continues creating exponentially more transistors and allowing exponentially greater design complexity. While it is desirable to create relatively homogeneous chips – simply stamping out a large array of simple designs – efficiency doesn't allow for this. Instead, designers are now creating accelerators, highly specialized hardware structures which provide extreme power, area, and time efficiency for a small range of algorithms. To motivate research in accelerators, some claim that a result of the power wall combined with Moore's law is "dark" silicon – area which is considered to be free, allowing designers to build a range of specialized units. The news in silicon economics, however, call this logic into question; the transistors are indeed not free – they are getting more expensive. However, accelerators provide radically higher performance at relatively low area. For example, a video decoder accelerator can provide nearly 3x the performance of a software implementation in more than 15x less area, leading to a 45x improvement in performance per area [65]. This in turn provides more value to the customer at lower cost to the producer, assuming the accelerator speeds up an application about which most customers care.

What the recent trend in accelerators may tell us is that chip companies are operating in a new regime: instead of profiting primarily through technology scaling, they may be looking to increase profits by extracting more value from each transistor. The goal of getting better performance from each transistor through accelerators, however, is at direct odds with increasing design cost. The more specialized a hardware design, the smaller the consumer market for the hardware, allowing less amortization of the design costs. The key, therefore, is to find accelerators which are either broadly useful or remain programmable enough to be broadly useful while still consuming little chip area. Rather than merely being a useful trait, the flexibility of accelerators will directly impact companies' bottom lines. We hypothesize this goal can only be achieved through strong quantitative understanding and analysis of applications and systems.

1.3 Overcoming the intuition wall

Both new and tradition pressures to create better, stronger, faster architectures motivate the use of quantitative methods to help understand applications and hardware systems. Informing intuition with quantitative understanding, however, is often not easy – sometimes due to the pains of collecting data, sometimes due to difficulties in attempting to analyze and understand these data. Ideally, when collecting data we measure values as semantically close to the desired values as possible. This often creates needs for new measurement tools. When the amount of data we can collect becomes too large for humans to manually analyze, machines can do a better job. This is especially true when data is high dimensional or otherwise complex. In these cases, manually inspecting data can be mis-informative. To overcome the intuition wall, we must further both measurement and analysis techniques. This dissertation presents four projects – two measurement techniques and two analysis techniques – which target both the traditional architecture application of program analysis and also a relatively new application area, security. Table 1.1 shows the breakdown of these projects.

	Measurement	Analysis
Program Analysis	**LiMiT** (Ch. 3)	Centrifuge (Ch. 5)
Security	SVF (Ch. 4)	Malware Detection (Ch. 6)

Table 1.1: The four new techniques and projects discussed in this dissertation

1.3.1 Measurement

LiMiT advances measurement using hardware performance counters [38]. Virtually all modern processors come equipped with counters than can monitor various architectural and microarchitectural events (like cache misses or branch mispredictions). These counters are most often used to find "hot" regions of code or regions that interact poorly with the microarchitecture for some reason. However, it is sometimes also useful to precisely study very small regions of code which may or may not run often. These regions may be interesting because they occasionally exhibit anomalous behavior or do not behave as one would expect. Existing tools, unfortunately, make precise, detailed measurement difficult as their overheads compromise accuracy. Instead, architects must often use very slow simulators (which may or may not accurately represent the microarchitecture) to take measurements. Forcing the use of simulators can add days, weeks, or sometimes even months to investigation of a problem. Often it is infeasible to run production workloads under simulation. With **LiMiT** we introduced a new method to conduct detailed study of programs using hardware performance counters. We also conducted several case studies of detailed program behavior revealing several unintuitive results about system behavior. Details on **LiMiT** can be found in Chapter 3.

SVF judges the security of a system with respect to information leakage using a new framework for creating metrics which we propose [40]. Side-channel information leakage occurs in systems where resources are shared – for instance in processors shared between multiple users. Software-level security mechanisms typically ensure that users cannot directly view each other's data, however side-channels can sometimes compromise these privacy guarantees. For instance, one very common side-channel attack is able to get information about bits in encryption keys used by OpenSSL by monitoring a shared cache. One particularly disturbing problem in side-channel defensive research is that there is no way to experimentally quantify leakage. Instead, designers must often intuit whether or not microarchictural changes increase or decrease leakage. For instance, does statically partitioning a cache always successfully eliminate information leakage? Intuitively, yes. Based on simple models of the processor and leakage, also yes. Based on experimentation and measurement with SVF, no. With SVF, we show reasoning about large, complex systems to be very difficult. In this area, a metric with which to quantitatively evaluate proposals and inform intuition is necessary. Further details on SVF can be found in Chapter 4.

1.3.2 Analysis

Centrifuge makes manual code analysis easier by scouring large code bases, searching for common patterns using a novel approximate graph clustering technique [39]. Its intent is find common patterns in large amounts of code which can guide designers. For instance, if architects are attempting to accelerate a particular set of applications, they could simply locate the most frequently run functions in those applications and design accelerators for each of those functions. This hardware, however, is extremely application specific and therefore area wasteful. Instead, the architect would like to design hardware that can accelerate multiple functions. However, not all functions are similar enough that an accelerator can handle them all. As such, it would be helpful to have a tool that finds pieces of code which may be similar enough to design a common accelerator for all of them. In Centrifuge, we developed and demonstrated algorithms to cluster functions using graph-based representations. Details on Centrifuge can be found in Chapter 5.

Performance Counter-based Malware Detection demonstrates the usage of relatively simple, lightly-guided machine learning algorithms in the detection of new malware [41]. Existing antivirus (AV) systems are very large, complex software systems which scan downloads and other new files, searching for matches in a database of known threats. Although there is much work in assisting the building of threat databases, the actual detection software is written manually and much manual forensic analysis of potential threats is still necessary. Unfortunately, the large amount of manually written code can be leveraged by malware authors to circumvent the antivirus system. For instance, all AV contain numerous parsers for many different file types – one for ZIP files, one for PDF, one for DOC, etc. Often times, these parsers contain bugs which cause them to interpret files differently than the software for which they are intended. Attackers can use these bugs to hide their malicious code from the AV system while it remains effective against the target application [80]. Instead of relying on humans to use their intuition in forensic analysis and skills in building large software systems, it may be preferable to have machines learn about malware threats themselves. In Chapter 6, we describe a system that uses supervised machine learning to let the machine build intuition about malware based on their microarchitectural behavior. Surprisingly, the machine is able to get relatively good accuracy using only simple learning methods.

None of these projects are end-all, be-all solutions to for quantitative methods. However, each one advances state of the art in a particular area of measurement or analysis: **LiMiT** allowed for higher accuracy, precise meaurement; SVF is the first holistic measurement methodology for side-channel information leaks; CENTRIFUGE introduced a novel method for mining databases of program graphs for interesting patterns; our malware detectors demonstrated the feasibility of automated malware protection and represents a possible new direction for a struggling field. Taken together, they argue for the utility of quantitative methods; they show that quantitative methods make research faster, less biased, and more insightful.

Chapter 2

Quantitative Methods in Computer Architecture

Computer architecture has long been a quantitative field: ideas are quantitatively evaluated, we use a variety of metrics to break down the operation of systems, and we use program analysis techniques to understand workloads [48, 66]. In fact, a huge variety of methodologies for measuring and analyzing hardware/software systems have been developed for a variety of different applications. In this chapter, we review some of this work, separated into applications and methods, as shown in Table 2.1.

Applications and the particular methods used by those applications are often conflated, intertwined as a result of their mutual development. Methods are developed for a particular application, they further that application and the resulting developments in that application space motivate improvements to the method. However, as evidenced by the use of phase analysis (a technique most often used for workload reduction) for security in Chapter 4, this need not always be the case. This chapter attempts to separate the two concepts whenever possible.

2.1 Applications

Nearly all ideas and designs in computer architecture are quantitatively evaluated and many are motivated by quantitative understanding of some software or hardware phenomena [48].

Applications		**Methods**	
Performance Power Reliability Security	**Evaluation**	Simulation Program instrumentation Hardware instrumentation Performance counting	**Measurement**
Locality, communication Control/data dependence Benchmark selection Workload reduction	**Program** **Comprehension**	Modeling Phase analysis Statistical analysis Compiler/code analysis	**Analysis**
Device modeling, trends μArchitectural Behavior	**HW** **Comprehension**		

Table 2.1: Select quantitative methods and some of their traditional applications

As a classic example, Chrysos and Emer [33] quantitatively showed the importance of memory dependence prediction, proposed a technique, and quantitatively evaluated it. In fact, this formula is extremely common in computer architecture papers: authors typically quantitively motivate a problem and provide some quantified intuition about the problem, introduce their solution and intuitively describe explain its potential, then quantitatively evalute their approach. Quantitative methods are involved in nearly all stages – comprehension of the problem to evaluation of the solution – to at least a small extent.

2.1.1 Evaluation

Performance, Power, Reliability Does a new technique or structure improve performance? Does it increase throughput? Reduce or increase latency? Increase energy efficiency and/or decrease power? How often will silicon fail, producing incorrect results? All of these are important questions for architects as they go about optimizing for performance, power, and reliability. Fortunately, these questions are quantitative and there exist metrics by which one can gauge these questions, allowing comparison between multiple approaches. Metrics like clocks per instruction (CPI) judge microarchitectural performance; operations per second judge system throughput; and energy per operation for energy-efficiency [48, 86]. Even reliability – an area with intuitively difficult to evaluate metrics like mean time to failure (MTTF) and mean time between failures (MTBF) [138] – is being characterized through works like Architectural Vulnerability Factor (AVF) [116].

Security Modern computing systems are constantly under attack from nefarious entities. As a result, it is increasingly important that computer systems to be resilient to attack. They must be capable of keeping private or secret information and use those data without leaks. Designers also need to ensure that systems cannot be controlled or disabled by unauthorized users.

Unfortunately, many sub-specialties within security research do not seem to have a history of being quantitatively evaluated. This is likely because security is a very difficult property define, let along evaluate quantitatively. Indeed, the likely best metric – "how difficult is it to break this system?" – seems inherently qualitative. As a result, security metrics often focus on detectable, countable events [28]: number of viruses are caught; number of break-ins occurred; the total cost of breaches; the level of compliance with security recommendations. However, many of these these metrics are not proactive and cannot be used to predict attacks or judge the improvement in security given by a new system. As a result, many new proposals must be judged intuitively and – likely as a result – are often broken quickly.

We surmise that the basic difference between judging security and other system properties is that security is adversarial. For example, when quantifying the reliability of a system with respect to soft errors caused by particle strikes, it is reasonable to assume a uniform probability distribution (or some other derived distribution) of strikes across the surface of the chip. After all, much as it may feel like it, the universe is not actively *trying* to make processors fail. However, in an adversarial situation this distribution cannot be assumed. Were an attacker attempting to subvert a processor's stability, he would purposely aim for the areas most vulnerable to attack. Thus, in order to soundly judge vulnerability one must provably find all the vulnerabilities – a difficult proposition as evidenced by decades of security research.

Despite this difficulty, there is a good deal of interest in finding ways to better quantify security [28, 81, 141]. Proposals are broadly broken down into two categories: experimental and formal. Experimental methods are able to judge the security of an actual system in situations as realistic as possible. As a result, however, it seems unlikely that experimental methods can be entirely robust – they could overlook aspects or untried attack vectors – so they may underestimate vulnerability. In contrast, formal metrics can provide mathematically sound upper bounds on vulnerability. However, they rely on being able to accurately model a system while retaining the ability to reason about it. As a result, formal methods are only valid with respect to a particular set of assumptions which are typically wrong in practice. For instance, cryptographic algorithms can be formally verified [88, 111], but can be subverted by attacks which violate their assumptions, like side-channel attacks [89–91, 156, 157].

2.1.2 Program & Hardware Comprehension

What does software do? What is hardware doing? How are the two interacting? These are the key questions in understanding the operation of hardware/software systems. Unfortunately, they are qualitative rather than quantitative in nature. As such, a host of characteristics which can be objectively discerned have been created to give quantitative answers.

Program Comprehension Often called program analysis (though not here, to avoid overloading the term with analysis methods), the ability to quantitatively study the operation of programs assists in optimization of programs and design of hardware for them. One of the most simple pieces of information is hot code (or hot spots) – what functions or lines of code are taking the most time on a CPU. These represent regions which are most important for optimization. However, despite being perhaps the most used application of program comprehension, hot spots are only a small portion of program comprehension. For instance: control and data flow graphs (*e.g.,* program dependence graphs [56]) have been some of the most important pieces of information about software since they directly govern instruction level parallelism. Additionally, high main memory latency makes the study of locality in programs (*e.g.,* via reuse distance [42] or Timekeeping [75]) important. Architects also examine the operation of programs at a coarser granularity through the study of program phases [134], allowing us to select smaller dynamic regions for study. Lastly, beyond understanding a single application, it is useful to compare different programs (typically running on the same microarchitecture) against each other; for instance, to select representative benchmark applications from a full set [129].

Hardware Comprehension Computers are large, complex, non-linear dynamical systems and the performance of software running on them is chaotic [16, 117]. As a result, minor changes to one component can have ripple effects throughout a system. Reasoning about and correctly predicting those effects can be impossible. As a result, quantitative methods are also necessary to fully understand the operation of hardware. The simplest are metrics for individual components: branch misprediction rate, cache miss rate, TLB miss rate, et cetera. With this set of metrics, one can often diagnose system bottlenecks ripe for optimization. There is also interest in the quantitative study of *how* microarchitectural structures operate rather than just *how well*. In security, there is work on formal modeling of cache systems for the purpose of measuring security [43]. The operation of two-level branch predictors has also been studied [52], and a reasonably accurate yet simple model for processor pipeline behavior is known [54] based on mechanical modeling.

2.2 Methods

To advance the goals outlined in the above applications section, computer architecture has created a large number of quantitative methods. Here we broadly break them down into measurement and analysis.

2.2.1 Measurement

It is often necessary to measure the dynamic characteristics of hardware/software systems. However, systems can be very difficult to measure accurately as a result of the uncertainty principle: systems are often disturbed by the act of measuring them, reducing the accuracy of the measurement. For instance, when making observations about the microarchitectural behavior of an application, one typically runs measurement software on the same system, which changes both the architectural and microarchitectural state, affecting the measurements. The result is that in computer architecture, measurement methodologies often face a trade-off between accuracy and precision (the difference between which is illustrated in

Figure 2.1). That is, in order to reduce the perturbation caused by measurements (and thus maintain high accuracy), architects must often use less precise methods like sampling.

Simulation One of the most popular tools in architecture are simulators. Beyond their usefulness in evaluating hardware without having to build it, simulators allow for measurements with infinite precision and the ability to measure any aspect of the system [48]. Simulation has two big problems, however. First, it is very slow – many orders of magnitude slower than the actual systems which they model. Second, simulators may or may not be accurate. Although they are designed to behave like the systems they are simulating, not all details of the system are captured. Some simulators (like Sniper [25]) explicitly eschew many system details in favor of lower simulation time. Other, cycle accurate simulators (like GEM5 [20]) attempt high accuracy, but the sheer complexity of modern systems (and proprietary nature of popular ones) limit their accuracy. Eeckhout [48] discusses the trade-offs in simulators between accuracy, simulator speed, development time, and coverage in depth.

Program Instrumentation For the study of software, an alternative to simulation is adding instrumentation code to applications. There are two standard methods: First, one can simply add code to the original application and recompile it. Second, there exist binary instrumentation tools [23, 101, 107] which allow one to insert code into the original

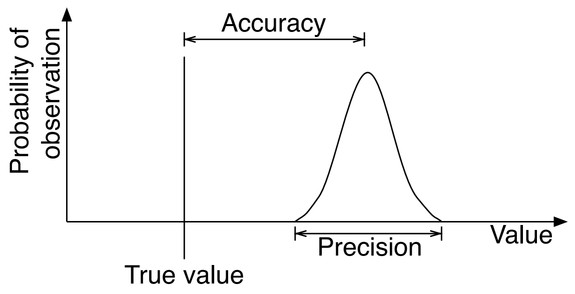


Figure 2.1: Although colloquially used interchangeably, accuracy and precision are different concepts. Accuracy refers to the difference between measured values and the true, hidden value which one is attempting to measure. Precision, however, refers to the potential variability in the measurement.

binary (either dynamically or statically), avoiding perturbation of the compilation process. Unfortunately, all of these methods significantly perturb the microarchitectural operation of software, so they can only be used to study software. Further, they slow down the program, so software which is affected by time cannot be studied without potential accuracy problems. Regardless of these issues, program instrumentation is useful for studying many dynamic behaviors: collection of control/data graphs; computing dynamic instruction mix; memory locality metrics like Reuse Distance [42] can even be collected.

Hardware Instrumentation It is sometime necessary to measure characteristics of hardware by physically instrumenting it. For instance, to measure the power dissipation of CPUs, a current sensor can be applied to its power inputs [51, 123]. More complex monitoring is also possible: the HMTT project inserts an FPGA between the processor and its memory, allowing high-speed configurable logic to monitor and record memory accesses [30]. Neither of these forms of instrumentation has an impact on the execution of software (though HMTT adds software instrumentation as well) so they have very high accuracy. The monolithic nature of modern integrated circuits, however, means that most hardware can only be instrumented at its periphery, limiting the usefulness of after-market hardware instrumentation.

Performance Counting Technically a form of hardware instrumentation, performance counters allow users to monitor a variety of architectural and microarchitectural events. All modern processor vendors have various performance counters; for instance, Intel provides counters for each core, for their "uncore", and for their integrated GPU. Since performance counters are able to monitor without perturbation, they have proven to be very useful. However, because the software to read their contents perturbs the system, they too have an accuracy-precision trade-off. We discuss performance counters further in Chapter 3.

2.2.2 Analysis

In order to deal with large amounts of data and inform ourselves better than by merely staring at data, there has been a huge amount of work in quantitative analysis for computer architecture. Since most quantitative analysis methods are designed for a specific application (of which there are many) we will discuss only select topics which are relevant to the projects in this dissertation with further details given in subsequent chapters.

Phase Analysis As mentioned in the applications section, programs tend to operate in phases: they do task A, then task B, then back to task A. The effects of these phases can be readily observed through architectural signatures (like basic block vectors) and microarchitectural behaviors. In short, specific phases tend to behave similarly each time they execute, making them interesting for study [70, 134]. There may be any number of ways to identify phases, though probably the most common is that of self-similarity matrices. One calculates these matrices from an input vector (often a time series) and compares each element to every other element with a distance function. These comparisons result in a matrix which highlights elements of the vector which are similar to others, revealing phases in the vector.

Machine Learning Computers can sometimes do a far better job learning from data than humans. As a result, there is interest using using machine learning to design microarchitectural structures. For instance, artificial neural networks have been used as branch predictors [82] and to manage resources [21]. Reinforcement learning [145] has been used to create better memory controller schedulers [77]. Since each microarchitectural structures is generally responsible for a well-defined set of actions given well-defined inputs and can often be optimized entirely based on a particular metric, it seems likely that many microarchitectural policies could be optimized by machine learning.

Compiler & Code Analysis Compilers have a large set of code analysis methods in order to support optimization; pointer analysis [71], for instance, is probably one of the most difficult and well known. More pertinently, however, there is also work which uses program graphs to automatically create small hardware accelerators: Clark *et al.* [34, 35, 72] use program mining to create custom instructions instruction set customization. Additionally, "Quasi-Specific cores" (QsCores) can be created to accelerate multiple regions of code by merging common parts of the code [153].

2.3 Conclusion

Computer architecture has benefited greatly over the years by using strongly quantitative methods, only a small number of which have been covered in this chapter. Despite the amount of work in this area, there is still much we do not understand and much further work in quantitative methods. For instance:

- Despite the rise of accelerator systems, quantitative methods regarding their design have yet to be fully discovered. Works like Clark *et al.* 's ISA customization [34, 35, 72] or QsCores [153] begin to answer the question, but only work for small, very specific accelerators, not the large, somewhat general purpose ones we would like to build. Chapter 5 speaks to this goal, providing another means of analyzing applications. In it, we introduce CENTRIFUGE, a program comprehension technique to help inform qualitative (intuitive) understanding of large amounts of code.

- There is very little work which examines hardware using black-box methods. Work like this would allow us to examine hardware and the (often complex) interactions between hardware components without bias. SVF in Chapter 4 takes this black box approach, introducing a framework for defining experimental security metrics.

- Measuring dynamic characteristics of systems is hindered by accuracy-precision trade-offs. Hardware like performance counters help alleviate the situation, but currently their usefulness is limited. Chapters 3 and 7.2 examine dynamic measurement further, discussing and advancing the use of hardware infrastructure to make precise and accurate measurements.

- Security – difficult as it is to quantify – begs for better methods by which to obtain unbiased evaluation and comprehension. We suspect that security defense as a field will advance little relative to the attackers without better quantitative methods. Two of the chapters in this dissertation (Chapters 4 and 6) examine aspects of security, the latter providing ways to measure security and the former analysis methods for malware detection.

Part I

Measurement

Chapter 3

Precise, Detailed, Real-Time Performance Measurement

On-chip performance counters play a vital role in computer architecture research due to their ability to quickly provide insights into application behaviors that are time consuming to characterize with traditional methods. The usefulness of modern performance counters, however, is limited by inefficient techniques used today to access them. Current access techniques rely on imprecise sampling or heavyweight kernel interaction forcing users to choose between precision or speed and thus restricting the use of performance counters hardware. This chapter redresses this key issue, introducing a new, fast method of accessing performance counters for precise measurement.

The contributations of this chapter are: (1) We describe new methods that enable precise, lightweight interfacing to on-chip performance counters. These low- overhead techniques allow precise reading of virtualized counters in low tens of nanoseconds, which is one to two orders of magnitude faster than current access techniques. (2) We use our new tool to provide several fresh insights on the behavior of modern parallel programs such as MySQL and Firefox, which were previously obscured (or impossible to obtain) by existing methods for characterization. (3) Based on several case studies with our new access methods, we discuss seven implications for computer architects in the cloud era and three methods for enhancing hardware counters further.

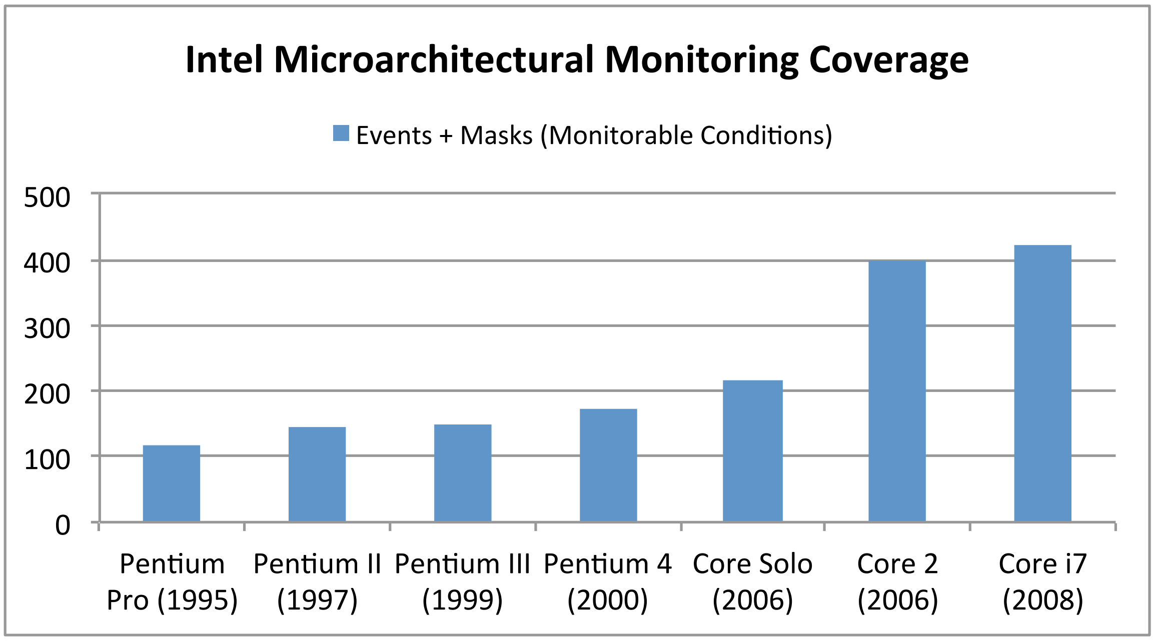


Figure 3.1: Number of countable conditions using Intel's performance monitoring framework through several generations.

3.1 Introduction

On-chip performance counters offer a convenient way to guide computer architecture researchers through the challenging, evolving application landscape. Performance counters measure microarchitectural events at native execution speed and can be used to identify bottlenecks in any real-world application. These bottlenecks can then be captured in microbenchmarks and used for detailed microarchitectural exploration through simulation.

Recently, some hardware vendors have increased coverage, accuracy and documentation of performance counters making them more useful than before. For instance, as shown in Figure 3.1, about 400 events can be monitored on a modern Intel chip, representing a three-fold increase in a little over a decade. Despite these improvements, it is still difficult to realize the full potential of hardware counters because the costly methods used to access these counters perturb program execution or trade overhead for loss in precision. We redress this key issue in this chapter with cheaper new access methods and illustrate how these methods enable observation of a range of new phenomena.

Popular tools used for accessing performance counters today such as PAPI [115], OProfile [120] or vTune [154] attempt to read performance counters via hardware interrupts or heavyweight kernel calls. An inherent downside of kernel calls is that they interrupt normal program execution and slow down the program thereby affecting the quantity being measured. To minimize these perturbations, most profilers resort to occasionally reading these counters and extrapolating full program statistics from the sampled measurements. While this extrapolation is necessarily imprecise, the error introduced by the process has been acceptable when profiling hotspots in serial programs.

Traditional sampling, however, has fundamental incompatibilities for parallel programs which have become commonplace with the availability of multicores. Traditional sampling methods are likely to miss small critical sections because they do not constitute the hottest regions of the code. Amdahl's law, however, teaches us that optimizing critical sections is necessary to ensure scalability, even if the time spent in critical sections is relatively low [53]. Moreover, as we will discuss in Sec 3.3.2, irrespective of the size, it is not easy to correctly monitor critical sections. Performance characterization of parallel programs with performance counters calls for simple, lightweight access methods that can enable *precise* performance measurement for both hot and cold code regions.

In this chapter, we describe novel lightweight techniques for accessing performance counters and report new application behaviors which are difficult to capture with existing access methods. Our precise access method, embodied in an x86-Linux tool called **LiMiT** (**Li**ghtweight **Mi**croarchitectural **T**oolkit), requires less than 12 *ns* per access (via 5 instructions) and is over 90x faster than PAPI-C [115] and 23x faster than Linux's perf_event, tools that provides similar functionality. **LiMiT** is the first tool to reduce precise counter reads to their minimal number of instructions while still ensuring correctness and virtualizing the counters across threads.

Based on three case studies with **LiMiT** using unscaled, production workloads we put forth several recommendations for architecture researchers.

In our first case study, we measure synchronization regions in production applications (Apache, MySQL and Firefox) as well as the PARSEC benchmark suite. Our measurements show that Firefox and MySQL spend nearly a third of the execution time in synchronization which is 10x more than the synchronization time in PARSEC benchmarks. These results indicate that synchronization is used differently in production system applications than traditionally-studied scientific/numerical applications and architects must be aware of these differences. Performing similar measurements with PAPI-C show inflated synchronization times due to high measurement overheads, drastically changed cycle count ratios and increased instrumentation overheads from 42% to over 745%. Some workloads such as Firefox could not even run properly with PAPI-C because of the high overheads.

Our next case study examines the interaction of programs with the Linux kernel via popular library calls. This interaction has not received much attention because of the difficulty in running modern, unscaled web workloads on full-system simulators. Our investigation reveals that production applications spend a significant fraction of execution cycles in dynamically linked libraries and operating system calls. Further, we find that routines in these two segments show distinctly different microarchitectural performance characteristics than userspace behavior.

The third and final case study demonstrates **LiMiT**'s breadth of utility by conducting longitudinal studies of modern software evolution. By examining the evolution of locking behaviors over several versions of MySQL, we investigate if there has been a return on investment in parallelizing the software for multicores. This study illustrates how the utility of precise counting goes beyond traditional applications in architecture, compilers and OS, and that well-architected performance counting systems can have wide and deep impact on several computer science disciplines.

3.2 Performance Counters Review

Performance counter based studies have proved exceedingly valuable in the past, and many influential research studies have been based on performance counter measurements of production systems. Emer and Clark shaped quantitative computer architecture with their seminal work on characterization of the VAX system using hardware counters [49]. Anderson *et al.* described results from system wide profiling on Alpha machines [6]. Ailamaki *et al.* describe results of profiling DBMS applications [2]. Keeton *et al.* characterized OLTP workloads on the Pentium Pro Machine [87]. Like these papers, we use novel performance measurement methods to study contemporary applications.

Performance counters started appearing in commercial machines in the '90s. The performance counter access facilities in these machines were intentionally minimalist to reduce chip area overheads. For instance, initial designs of the Alpha 21064, one of the first machines to include performance counters, did not even have read/write access to the performance counters. To keep area overhead low, the counters interrupted processor execution when a counter overflowed, allowing only basic sampling support based on interrupts [135]. As the usefulness of the counters became clear and transistors became cheaper, later Alpha chips and other vendors' chips enhanced their performance counter infrastructure. By the late '90s, all of the major processor lines, including Pentium, PPC, UltraSparc, PA-RISC and MIPS processors included performance counters and simple access methods.

A common feature of many of the counter designs in early processors – and a source of major frustration to date – is that all of these counters were accessible only in the privileged mode, thus requiring a high overhead kernel call for access. This problem was mitigated to an extent in the MIPS R10000 [161] (1995), which included support for both user-/kernel-level access to the performance counters. Later x86 machines from Intel and AMD have included similar configurable support. However, the software used to access the counters (kernel and libraries) often do not enable user space counter reads by default, likely to allow them to mask the complexity of counter virtualization behind the kernel interface. A recent proposal from AMD [5] published in 2007, discusses lightweight, configurable user space access. The proposed scheme appears promising but hardware implementations are not yet available.

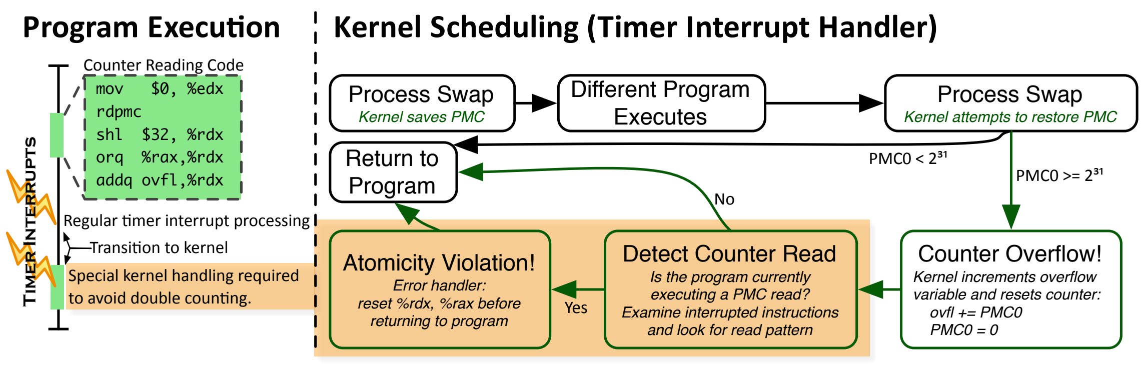
Hand in hand with the hardware improvements, many software tools have been developed over the years to obtain information from performance counters. These tools can either pull data from the performance counters on demand (precise methods) at predetermined points in the program or operate upon data pushed by the performance counter (imprecise methods) during externally triggered sampling interrupts. Intel's vTune [154] and DCPI/ProfileMe [37] are some commercial examples of tools that support only imprecise access methods. An open source example is the Performance API (PAPI) which was created in 1999 to provide a standard interface to performance counters on different machines [115]. OProfile [120] is another Linux profiling tool that provides interrupt-based sampling support. With these tools, users can extrapolate measurements obtained from samples collected either at predetermined points in the program or during sampling interrupts triggered by user specified conditions, *e.g.,* N cache misses. A general drawback to these sampling methods is that it introduces error inversely proportional to the sampling frequency. As a result, short or cold regions of interest are difficult to measure precisely.

Examples of tools that provide precise performance monitoring access methods for Linux are perfmon2 [127], perf_event [58] and Rabbit [130]. Perfmon2 is an older Linux kernel interface which provides both sampling support and precise counter reads, though the precise read support requires system calls. The newly introduced perf_event interface is intended to replace perfmon2 but still uses system calls (the read syscall, specifically) for precise access to performance counters. Rabbit is an older access method written to avoid system calls, but provides none of the virtualization features of **LiMiT**, perfmon2 or perf_event.

With the exception of Rabbit, all these tools require that performance counters be read by the kernel, requiring heavyweight system calls to obtain precise measurements. Unlike the above tools, our access techniques provide both precise and low overhead measurements by allowing userspace counter access. We compare our measurements to PAPI-C and perf_event, showing that by enabling userspace access, **LiMiT** introduces less perturbation than PAPI, and decreased overheads enable accurate, precise profiling of long running or interactive production applications.

3.3 Enabling Low-overhead Performance Counter Access

In this section, we describe the techniques used to reduce counter polling overheads and compare the overheads of our technique to existing alternatives. To summarize, our technique moves counter reading from kernel space to user space, creating issues which can occasionally cause invalid counter reads. We these issues correctness violation detection and elision which only have to run during process context swaps, so virtually no overhead is added. The result of our techniques is 23x speedup over existing functionally equivalent



```
mov   $0, %edx
rdpmc
shl   $32, %rdx
orq   %rax,%rdx
addq ovfl,%rdx
```

Program Execution
Counter Reading Code
TIMER INTERRUPTS
Regular timer interrupt processing
Transition to kernel
Special kernel handling required to avoid double counting.
Kernel Scheduling (Timer Interrupt Handler)
Process Swap
Kernel saves PMC
Different Program Executes
Process Swap
Kernel attempts to restore PMC
PMC0 < 2^31
PMC0 >= 2^31
Return to Program
No
Atomicity Violation!
Error handler: reset %rdx, %rax before returning to program
Yes
Detect Counter Read
Is the program currently executing a PMC read? Examine interrupted instructions and look for read pattern
Counter Overflow!
Kernel increments overflow variable and resets counter: ovfl += PMC0 PMC0 = 0

Figure 3.2: LHS figure shows **LiMiT**'s five instruction counter read sequence (dotted box) embedded as part of regular program execution. As shown, program execution can be interrupted when the program is executing uninstrumented code or when executing user space code for reading counters. Interrupts received during counter reads require special handling to avoid double counting bugs. RHS figure shows special modifications (highlighted boxes) that provide detection of interrupted counter reads and fixes for double counting bugs.

software while maintaining important features like process isolation.

3.3.1 Enabling Low-overhead User Space Access

Enabling user space access is a three step process:

• **1:** Stock Linux kernels do not allow direct user space access to performance counters. As a simple first step, we set the configuration bit (an MSR in x86) to allow user access.

• **2:** Performance counters cannot be directly configured to monitor events of interest (*e.g.,* instructions retired) from user space. We add a system call to the Linux kernel to configure the counters. Since most applications are likely to set up these counters once or few times per program we do not take any special measures to optimize this step.

• **3:** A more involved third step is to enable process isolation by virtualizing the operation of the performance counter hardware, allowing multiple programs to use one hardware instance of the performance counters. Without this support, programs would read events which occurred while other programs were executing, resulting in incorrect results and also opening up side-channels that can be used to infer information about program execution.

In theory, virtualization support should be as simple saving and restoring the performance counters during context swaps just like any other register. However, we need to deal with the possibility of performance counters overflowing. Intel's 48 bit counters can overflow every 26 hours, so overflows are likely for long running applications. Additionally, Intel chips prior to Sandy Bridge allowed only 32 bit writes to the counters so after only 1.4 seconds the kernel can find itself unable to correctly restore the counter when a process is swapped back in.

We work around overflows by detecting overflow conditions and accumulating the overflowed values in user memory. When a process wants to read a performance counter it must get the current value via `rdpmc` then fetch and add the contents of the overflow value in memory. However, this set of instructions must be executed atomically; if an interrupt and overflow occurs during their processing (before the memory fetch but after the `rdpmc`) then the value read will be off by the previous value of the counter as the kernel has zeroed the already read counter register and incremented the as-yet-unread overflow variable.

Two obvious solutions to ensure atomic execution, turning off interrupts or protecting the critical section with a lock, cannot work in this context. If interrupts are disabled, the executing process would never be swapped out and could starve other applications; allowing a user process to disable external interruption is dangerous. Locking is even more problematic. Our algorithm requires the kernel to update the user space memory location that keeps track of the performance counter values. To do this the kernel must obtain a lock when the process is being swapped back in. However, if the process holds the lock, then the kernel cannot continue and the process will never resume to release the lock. In this situation deadlock is guaranteed.

Linux kernel interfaces such as Perfmon2 and perf_event deal with this problem by placing all sensitive code in the kernel where techniques like disabling interrupts can operate normally. By doing so, however, they add significant overhead to counter reads in the form of system calls to access counters.

To solve this problem, we use an approach similar to Bershad *et al.* [17] (Figure 3.2). We speculatively assume that there will be no atomicity violation, but build detection and error handling into the kernel code for cases where such events happen. With this approach, there is no additional overhead added to counter reading code in user space and overhead is only incurred on relatively infrequent counter overflows. To detect whether or not an application is in the middle of a counter read during a counter overflow we simply check the pattern of instructions before the process was interrupted (pointed to by the process' instruction pointer). If a counter read is detected, the kernel zeros the process' registers (%rax and %rdx in the x86 example) to match the new (overflowed) contents of the performance counter. Once resumed, the program will behave as if the interrupt, context switch and overflow had occurred immediately prior to the read of the performance counter. The primary difference from the approach in Bershad *et al.* [17] is that they rewind execution to the beginning of the critical section instead of fixing up the correct counter values as we do.

3.3.2 Comparison to Sampling

Sampling is typically used in two ways: interrupt based or by polling. In interrupt based sampling, interrupts are triggered when a pre-determined event such as number of committed instructions reaches a pre-determined count. These interrupts are received by the OS and passed on to the application. In polling based sampling, the counters are precisely read out once out of every N times a code region is executed to reduce overhead. While both approaches can have low overheads, there are a number of situations in which neither approach works well.

For example, Figure 3.3 contains a critical section from MySQL which accounts for 30% of MySQL's overall critical section time. Let us say that we are interested in measuring time spent in critical sections using interrupt based sampling. If K of the N samples were in critical section we would extrapolate that K/N of the total time was spent in critical sections. However, there are several complications with this approach. In the above example, a sampling interrupt routine which fires during the critical section, would have difficultly determining whether or not a lock is held because the locks are executed based on the `if` conditional preceding the lock.

An alternative to interrupt sampling is to use precise access methods intermittently. In this case, explicit performance counter reads would have be used every time a lock is

```
if (info->s->concurrent_insert)
  rw_rdlock(&info->s->
              key_root_lock[inx]);

changed=_mi_test_if_changed(info);
if (!flag) {
  switch(info->s->
         keyinfo[inx].key_alg) {
  /* 37 lines omitted */
}
if (info->s->concurrent_insert) {
  if (!error) {
    while (...) {
       /* 10 lines omitted */
    }
  }
  rw_unlock(&info->s->
            key_root_lock[inx]);
}
```

▨ Conditional Locks

Figure 3.3: Code excerpt from MySQL 5.0.89, `mi_rnext.c`. The critical section shown here accounts for 30% of all the time spent in critical sections.

acquired or released. To reduce overhead, performance counter reads could execute only once out of every N times the region is entered, and the total time could be extrapolated from this measurement. While this method is effective in reducing overall overhead, the overheads for each precise read remain high. As a result, large perturbation is introduced immediately before and after the region of interest when measurement is actually occurring. We would therefore expect measurements for small regions to be inflated. We observe this effect during our Case Study A in Figure 3.5b.

In many of these situations in which sampling or heavyweight precision present difficulties, *ad hoc* solutions are possible. However as our case studies demonstrate, a low-overhead, precise measurement like **LiMiT** is sometimes the right tool for the job.

3.3.3 Comparison to PAPI and perf_event

For years, PAPI has been the standard library to write cross platform performance monitoring tools. As a library, it relies on kernel interface support; traditionally it has used perfmon2 on Linux. In contrast, perf_event is the newest Linux kernel interface. It is touted to be faster and more featureful than perfmon2 and will thus eventually replace it. However,

Time	PAPI-C	perf_event	**LiMiT**	Speedups	
User	1.26s	0.53s	0.34s	3.7x	1.56x
Kernel	30.10 s	7.30s	0s	∞	∞
Wall	31.44s	7.87s	0.34s	92x	23.1x

Table 3.1: Speedups of **LiMiT**, perf_event, and PAPI (10^7 reads of 3 counters) plus **LiMiT**'s speedup over PAPI and perf_event respectively.

due to its relative youth, library support for perf_event remains poor, placing burden on the user but yielding better speeds as there is no library overhead.

Any performance counter readout call (be it PAPI or **LiMiT**) will cost some number of cycles. To examine this overhead, we construct a short benchmark which reads a counter configured to count three events (cycles, branches and branch misses) 10^7 times each. With this high number of iterations, we can report the wall time for comparison of the overheads and compute the cost of each readout call. The results are presented in Table 3.1. On our Xeon 5550-based system, the average for **LiMiT**'s five instruction readout code is 37.14 cycles. Since **LiMiT** does not require a system call for each sample, it is substantially faster compared to PAPI-C (by 92x) and perf_event (by 23x).

In Section 3.4, we instrument MySQL to examine locking, unlocking and critical section timing (setup described in detail in the following section). Figure 3.5b shows that using **LiMiT** incurs a 42% cycle increase over uninstrumented execution. When the same instrumentation is performed using PAPI, a 745% user space cycle overhead is introduced and 97% is incurred with perf_event. Both PAPI's and perf_event's actual overheads, however, are much larger since over 90% of their overheads occur in kernel space (as shown in Table 3.1) but are not counted in figure 3.5b. As a result, we would expect both PAPI and perf_event instrumentation to perturb execution more than **LiMiT** making the results virtually unusable.

Overheads also directly affect usability. We attempted to instrument and measure modern cloud workloads such as Firefox, MySQL and Apache with both **LiMiT** and PAPI. Firefox was unresponsive to input with PAPI, while it operated with no discernible slowdown when instrumented with **LiMiT**. We also measured that Apache served 9,246 requests per second with **LiMiT** instrumentation and 9,276 requests per second without instrumentation. These minor changes in speed demonstrate **LiMiT**'s low overhead.

3.3.4 Comparison to RDTSC Measurements

Using `rdtsc`, the read time stamp counter instruction on x86 architectures, is *de rigeur* in userspace lightweight measurement. The time stamp counter is a free running counter present on all x86 machines. It simply counts bus cycles (uncore cycles for modern Intel processors) and most operating systems allow programs direct access to it. Since `rdtsc` is simple and lightweight, programmers will often use it to measure the time spent in short or long regions of code or to judge the effect of code changes on performance. **LiMiT**, however, offers capabilities that are superior to plain `rdtsc`: aside from offering a variety of countable events besides bus cycles, **LiMiT** provides process isolation which allows each process to shield its measurements from other processes' direct interference. While one could apply many of **LiMiT**'s techniques to `rdtsc`, this does not occur in practice so we compare against `rdtsc` without any such additions.

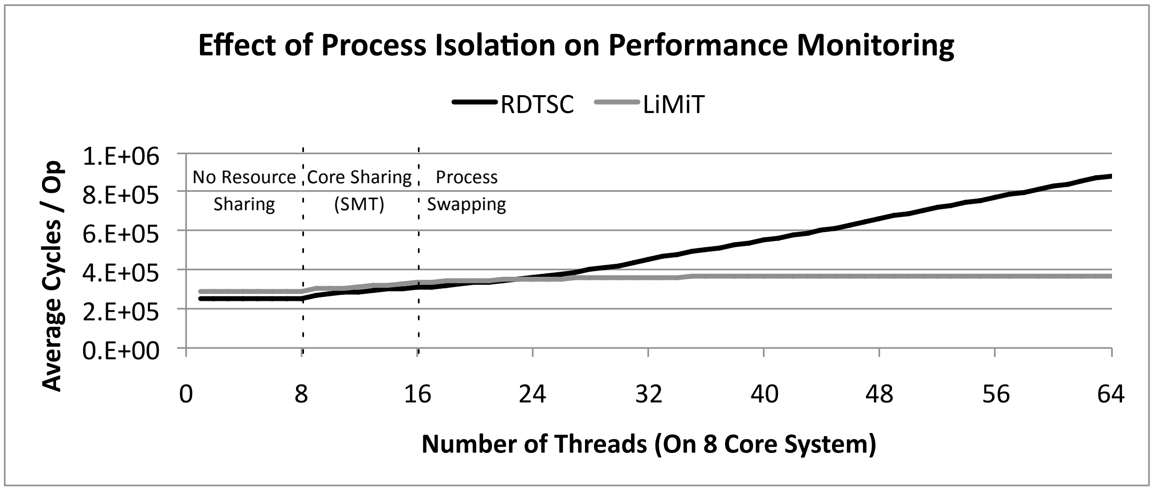
To examine the effect of process isolation, we construct a simple microbenchmark which executes non-memory operations across multiple threads on an 8-core system, allowing the operating system to schedule them onto cores. We then compute the average amount of time each operation takes using both `rdtsc` and **LiMiT**. We would expect the performance of each operation to degrade as resource sharing increases. There should be little or no performance degradation with 8 or fewer threads, mild degradation from 8 to 16 threads as SMT is utilized then a little more performance degradation above 16 threads as threads are swapped in and out. The data presented in Figure 3.4b confirm these expectations when using **LiMiT**. `rdtsc`, however, incorrectly reports massive, linearly increasing performance degradation above 16 threads as a result of its lack of process isolation.

3.4 Case Study A: Locking in Web Workloads

Usage patterns of computers have changed drastically over the past decade. Modern computer users live in the cloud. These users spend most of the their time in web browsers

```
#define rdtsc(X)              \
asm volatile ("rdtsc;"        \
     "shl $32, %%rdx;"        \
     "orq %%rax, %%rdx;"      \
   : "=d"(X) :  : "%rax");

int main(void) {
  uint64_t b, e;
  rdtsc(b);
  for (uint64_t i=0;
       i<ITER; i++) {
    // ... some operation
  }
  rdtsc(e)
  printf("Time per op: %lf\n",
   ((double)e - b)/ITER);
}
```

(a) RDTSC Example



(b) RDTSC Isolation Effects

Figure 3.4: Top: typical `rdtsc` usage example. Bottom: Process isolation in **LiMiT** prevents other threads and processes from directly affecting event counts. RDTSC has no such ability.

– either on a traditional desktop or mobile device – rather than in native applications, which moves computation to backend servers. As a result, there are two separate and extremely important workloads in the web model: the frontend, consisting of web browsers and Javascript engines, and the backend, consisting of HTTP servers, script interpreters and database engines. Further, the workloads of these applications have also changed. Often web pages rely far more on Javascript than ever before and database operations are no longer well modeled by traditional transactional benchmarks, often favoring scalability and speed over data security and transactional atomicity and durability.

We briefly characterize the synchronization behavior of several popular web technologies. Specifically, this study aims to answer the following questions: (1) Is synchronization a concern in web workloads and what are the locking usage patterns? (2) What future architecture directions can optimize web workloads? For comparison purposes, we also measure and analyze the PARSEC benchmark [18]. As a numerical workload, PARSEC is more representative of traditional (scientific computing) notions of parallel programming and may be different from web technologies.

Necessity of LiMiT There are three features offered by **LiMiT** which enable this study: precise instrumentation, process isolation and low-overhead reads, not all of which are simultaneously offered by other technologies. Precision is necessary because we are capturing very short regions of executions – lock acquires/releases and critical sections – which are likely to be missed by sampling techniques. Process isolation (which is not offered by the traditional `rdtsc`) is required since we are operating in a multi-threaded environment with I/O, so processes are likely to be swapped in and out often. Finally, **LiMiT**'s low-overhead counter readout routine is required to prevent large perturbation from skewing results. To further examine **LiMiT**'s lowered overhead, we will compare results obtained with **LiMiT** to results obtained with PAPI.

Experimental Setup To gain insight into modern web workloads, we examine the following software and input sets:

Firefox A popular, open-source web browser, we ran Mozilla Firefox version 3.6.8. We visited and interacted with the top 15 most visited sites, as ranked by Alexa. Additionally, we used two web apps from Google, Gmail and Google Reader[1], two applications which rely heavily on AJAX, asynchronous Javascript and XML.

Apache The Apache HTTP server is, according to Netcraft, the most popular HTTP sever with 56% market share as of August 2010. We evaluated the latest stable version, 2.2.16, using the included "ab" (Apache Benchmark) tool to fetch a simple static page. A total of 250k requests were served with 256 requests being requested concurrently. Because we look only at static loads, the results will indicate a best-case scenario for Apache.

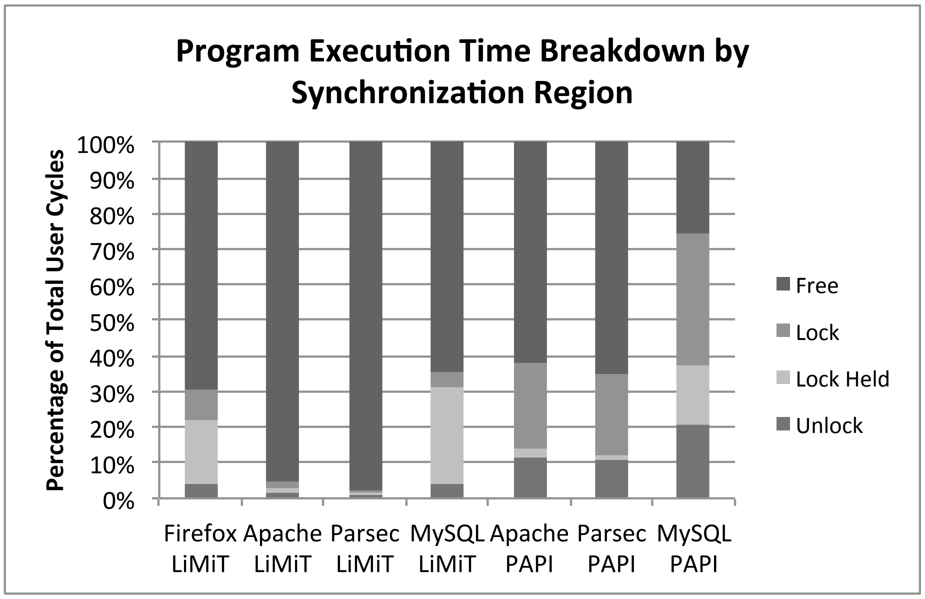
MySQL MySQL is the traditional database server of choice for websites. The most recent stable version is MySQL 5.1.50 Community Server, which we evaluated. To exercise its functionality, we ran the "sql-bench" benchmarking scripts included with MySQL's source code.

PARSEC The PARSEC benchmark suite [18] is a set of parallel applications largely targeting RMS workloads. We executed seven of the multithreaded benchmarks: blackscholes, swaptions, fluidanimate, vips, x264, canneal and streamcluster.

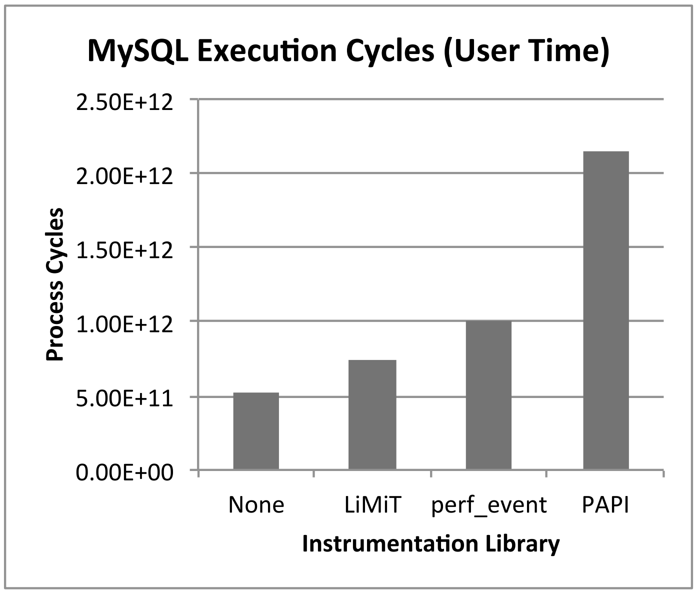
We instrumented each of these applications using **LiMiT** to track their critical sections and locking behaviors. Specifically, we collected information on the number of cycles spent acquiring and releasing locks, and time spent with locks held.

Results The charts in Figures 3.5 and 3.6 summarize the collected data. Figure 3.5 contains an overview of synchronization overheads and critical section times. Execution time is computed as the total number of cycles in all threads, lock and unlocking times as all time spent in `pthread_mutex_lock` and `pthread_mutex_unlock` in all threads. Lock held time, however, is defined as summation of the amount of time each thread has at least one lock held; if more than one lock is held, time is not double-counted.

[1]Since the time of experimentation, Google Reader has been discontinued. It was a cloud-based RSS aggregator.



(a) Synchronization overheads



(b) MySQL cycles counts

Figure 3.5: Comparison of synchronization and critical section timing for various popular applications and the PARSEC benchmark suite along with execution times for MySQL. Results obtained with PAPI are inflated due to instrumentation overheads. We also see that PAPI instrumentation increases userspace cycle counts by more than 745% compared to **LiMiT**'s 42% increase. We also note that Firefox (being an interactive program) could not execute with PAPI instrumentation.

	Firefox	Apache	PARSEC	MySQL
Average Lock Held Time	789	149	118	1076
Dynamic Locks per 10k Cycles	3.24	1.12	0.545	3.18
Static Locks per Thread per Application	57	1	17	13853

Table 3.2: Locking-related averages. We note that the vast majority of PARSEC's static locks are observed in one benchmark: fluidanimate. Without this benchmark, the number of static locks per thread per application drops to 0.575. These data indicate that scientific and web workloads have significant difference in synchronization behavior.

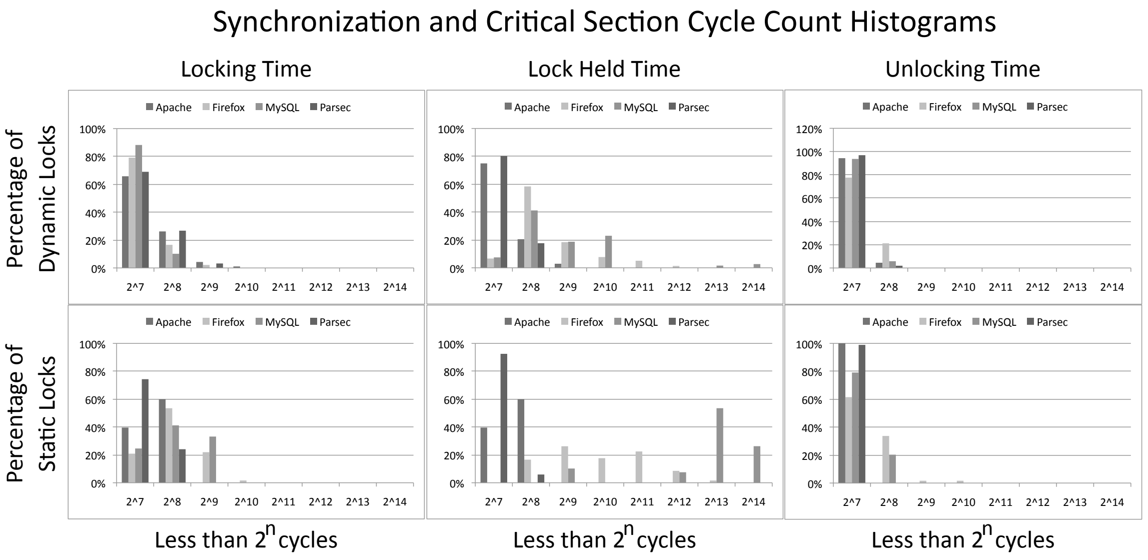


Figure 3.6: Histograms of synchronization overheads and critical section times for several applications. Times are broken down by dynamic locks (number of lock acquisitions) and average for each static lock (observed lock instance). We note that many critical section times are very short, comparable in cycle counts to lock acquisition times.

These data show that locking behavior varies a great deal between the applications. Figure 3.6 contain histograms of locking and unlocking overheads (latency of lock acquire and release) and times spent in critical sections. We break down this data by both dynamic locks (number of lock acquires during execution) and static locks (number of lock instances observed during execution), revealing insights about lock usage patterns. From this data, we make several observations:

Critical Section Times The histograms in Figure 3.6 indicate that the manner in which each application uses locks varies. PARSEC, for instance, holds locks for very short amounts of time in stark contrast to MySQL and Firefox. (See Table 3.2.) This is likely because many of PARSEC's applications parallelize nicely, *e.g.,* using data parallelism and static assignment. The other applications, however, are interactive and must respond to events as they occur. Since this makes static assignment impossible, threads must interact more often, requiring more synchronization.

Number of Locks The previous point is further supported by the number of locks shown in Table 3.2. Highly interactive applications like Firefox and MySQL require significantly higher number of locks. PARSEC is likely able to use only barrier-like constructs to synchronize computation.

Based on this data, we will attempt to answer the questions set forth. To answer our first question, about locking patterns in web workloads, we observe that synchronization is a mixed bag in web applications. Some workloads, like Apache, are likely to be very parallel and scale easily. MySQL does not fit into this category as it does not scale as easily. Additionally, Firefox has far more synchronization overheads then one would expect. Based on personal experience with Mozilla code, we suspect this is a result of difficulties in parallelizing legacy "spaghetti" code which is likely to have many side effects which must be isolated from other threads.

Implications for Architects (#1, #2, #3) Our second question — How are architects affected by these results and what future directions would best support the web? — bears further analysis. There are several interesting points:

# 1: **New Benchmarks** A new benchmark suite of web software may be necessary for new web-centric architecture research. SPEC has several versions of the "SPECweb" benchmark; future studies should include comparisons. However, many of the applications we have reviewed and other important cloud workloads are not part of SPECweb, including Firefox, Javascript, website supporting databases (non-transactional workloads), server caching and load balancing.

# 2: **Locking Overheads** Our data show locking overheads can be non- trivial compared to critical section times. Since locking/unlocking overheads can be 8% to 13% of overall cycles, speedups in this range may be possible with architectural/software techniques for streamlining lock acquisition. Further, we observe that the static lock distributions differ from the dynamic lock distributions, suggesting that one may be able to statically determine which locks are likely to be contended and which are likely to be held for many cycles.

# 3: **Critical Section Serial Performance** Critical section times for MySQL are relatively large. In particular, over half of the lock instances have average lock hold times around 8,000 cycles (although they are locked less often). These represent segments of code which will not scale well. These regions are prime targets for microarchitectural optimization. If they can be sped up, parallel performance and scalability of MySQL will improve.

3.5 Case Study B: Kernel/Userspace Overheads in Runtime Library

Our next case study is aimed at examining the interaction of programs with the Linux kernel via popular library calls and understanding their impact on program performance. A prior study has shown that kernel calls can negatively impact performance by polluting branch predictors [104]. Are there other on-chip structures that are affected by kernel calls? To what degree are modern applications affected by their kernel interaction? Is it possible to obtain fine-grained information about execution that can be tracked back to originating function calls? Our goal is to use **LiMiT** to study common library functions' behaviors in both userspace and kernel space.

Necessity of LiMiT There are two alternatives to using **LiMiT** for collecting this data.

First, simulation can be used to study the interaction of user and kernel code. Full system multiprocessor simulators can model the effect of system interaction and can shed light on effect of library calls but can be prohibitively slow without scaling workloads. Although **LiMiT** cannot achieve the accuracy and detail level of simulation, it can be used to rapidly gather precise information and coarsely locate problem regions.

The second option is sampling with external interrupts. This style of sampling provides an interrupt every N events at which point the sampling interrupt can analyze the application's execution state. In this study, however, we must determine which library functions use processor resources *and* the purpose of the function calls. For instance, we would like to know whether `memcpy` is manipulating program data or copying data for I/O. Obtaining this data in both user and kernel space is difficult for sampling-based methods as each sample interrupt must also run a stack trace (often from the kernel stack all the way back to and through the user stack) to identify the library entry point. To our knowledge, no existing sampling tool is able to track kernel function usage back to the calling userspace function. While theoretically possible for sampling, **LiMiT** makes this approach downright easy. With **LiMiT**, we read counters at the entry and exit points of functions in each category, so all events occurring between the function entry and exit, including all functions called from within the function, are counted towards that function. For example, if `pwrite` calls `memcpy` internally or the kernel executes some locking functions during a `read` system call, any microarchitectural events resulting from the `memcpy` or kernel locking will count towards `pwrite` or `read` rather than memory or locking categories.

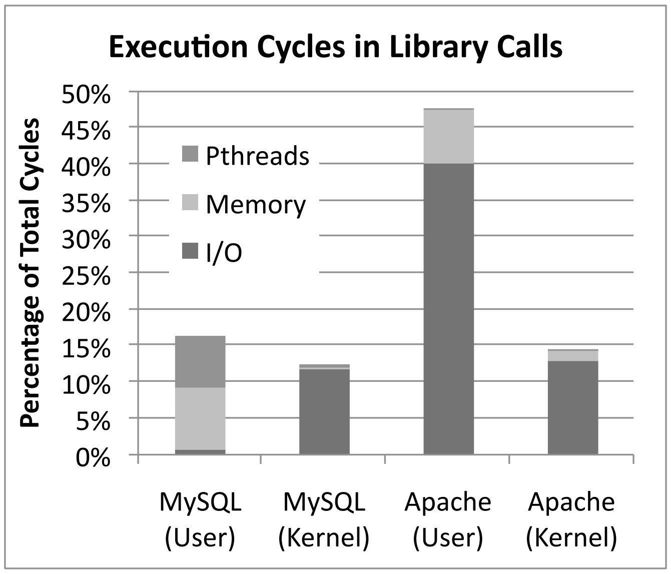
Experimental Setup To examine the effects of kernel code, we intercept and instrument functions in libc and pthreads. During calls to these libraries, we count cycles, L3 cache misses and instruction cache stalls in user space and kernel space separately. After collecting data, we aggregate the data from each function into three separate categories: I/O, memory and pthreads. I/O contains functions such as `read`, `write` and `printf` whereas memory has functions like `malloc` and `memset`. Pthreads contains all of the commonly-used synchronization functions. We look at two important systems applications, Apache and MySQL, using the workloads described in Section 3.4.

Results The results of this study are shown in Figures 3.7, 3.8 and 3.9. Figure 3.7 reveals potential inefficiencies. First, we observe that MySQL spends over 10% of its execution cycles in kernel I/O functions. Apache spends a comparable amount of time, but also spends a large amount of time in user I/O code. Overall, in fact, Apache spends the majority (about 61%) of its cycles in library code. Looking at cache information, Figure 3.7b shows that kernel I/O experiences far more cache misses per kiloinstruction than userspace code. The last chart, Figure 3.7c helps explains further, revealing extremely poor instruction cache utilization in kernel mode, especially in I/O functions.

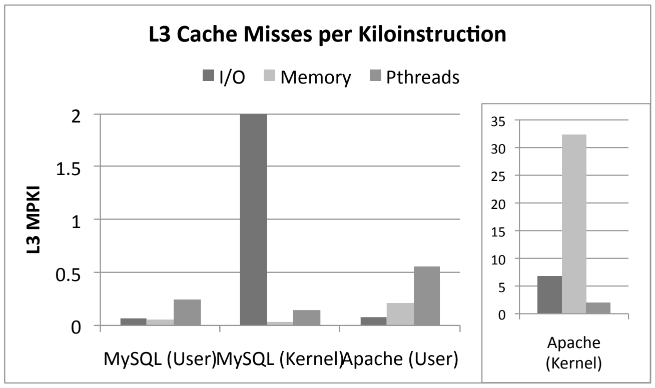
Figures 3.8 and 3.9 show the CPI and last level cache misses for the worst performing functions in libc plus aggregates of userspace code, kernel code, library functions and normal program code. These data show that kernel code does not perform as well as userland code and that several functions perform very poorly, especially in terms of cache misses. In particular, the math function `floor` performs very poorly (due largely to cache misses) though it does not contain a kernel call. Fortunately, MySQL does not call it often (241 times compared with 4.4e8 times for memcpy). The infrequent calls and last level cache miss results suggest that that poor temporal locality and prefetching of mathematical constants or code in libm may be to blame for the poor performance.

Implications for Architects (#4,#5,#6) The first important result from this data is that system applications have large amounts of kernel interaction and their behavior in kernel regions is markedly different from userspace. As a result, userspace-only simulation misses potentially important information. Additionally, there are two key observations in the above data which indicate potential avenues for optimization:

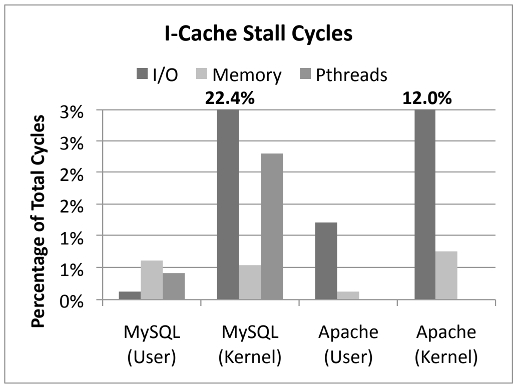
# 4: I/O Optimizations The Apache results show the importance of I/O optimization. Apache spends much time interacting with the kernel, incurring significant overheads. Hardware support to allow Apache (and similar programs) to circumvent the kernel



(a) Cycles in Library Functions



(b) Last Level Cache Misses



(c) ICache Stalls

Figure 3.7: Various user space and kernel space microarchitectural events occuring in categories of library functions. Comparing userspace to kernel, we see that kernel code behaves very differently than userspace code. Please note the different scale in (b) for Apache in kernel space.

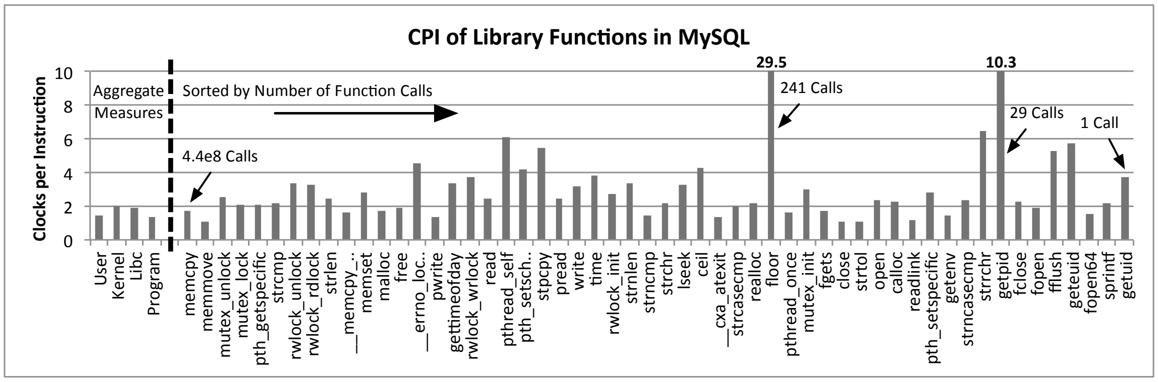


Figure 3.8: Cycles per instruction for various library functions executed by MySQL are listed here, sorted by number of calls. We see that in many cases, code in the dynamically linked library performs worse than typical program code. The same is true of kernel code to an even greater extent. Although performance is particularly poor for functions like `floor` and `getpid`, they are not called often and thus do not affect overall speed.

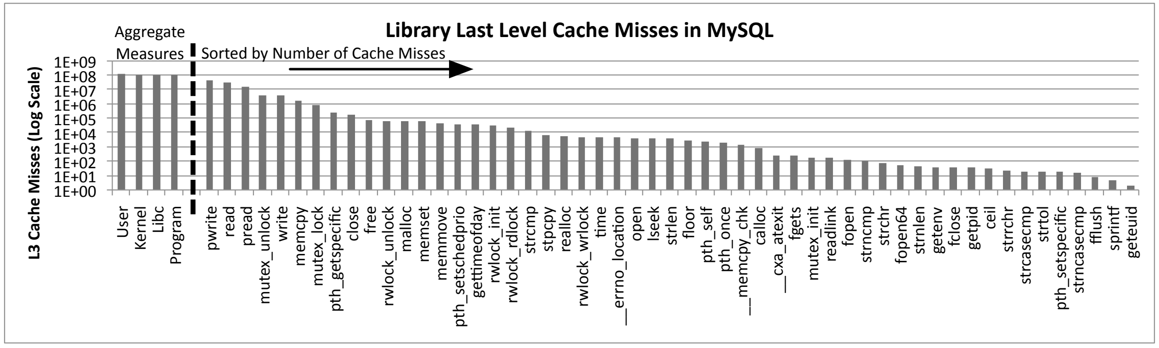


Figure 3.9: L3 cache misses in various dynamically linked library functions show that a handful of library functions account for a large portion of all the cache misses. Many of these functions result in kernel calls which suffer from abnormally high cache miss rates, as seen in Figure 3.7b. The MySQL benchmark executed for these data uses a database growing up to 45MB in size, relative to 8MB of CPU cache.

to do its I/O could drastically decrease its latency and increase throughput.

# 5: **Syscall I-Cache** Poor instruction cache behavior in kernel mode may indicate that the processor is unable to prefetch kernel instructions before interrupts occur. It should be possible for a hardware prefetcher to determine the system call number and prefetch the necessary upcoming instruction code, avoiding I-Cache misses.

# 6: **New Research Style** Finally, this **LiMiT**-obtained data has identified several problem points in real applications with unscaled workloads. With **LiMiT**, a process that would have taken months using simulators took only 3 days. If microbenchmarks can be designed to capture these bottlenecks, they can be used in full system simulation. This style of combining **LiMiT**'s precise event counter approach with detailed simulation may be necessary for quantitative architecture research in the cloud era.

3.6 Case Study C: Longitudinal Study of Locking Behavior in MySQL

Embarking on parallelization is often a risky investment with little guarantee of performance improvements due to the difficulties in writing multithreaded code. Many organizations that have legacy sequential codes are hesitant to invest in parallelization without quantitative models that can be used to predict return of investment on parallelization. **LiMiT** offers capabilities to build such a model.

In this case study, we use **LiMiT** to examine the benefits of adapting software to multicores over multiple versions spanning years. To examine software development progress, we examine several versions of MySQL, an extremely popular database management system. Gartner Group estimates that 50% of IT organizations had MySQL deployments in 2008, making MySQL a very common workload. As an open source product, we are also able to access its source code from many versions going back to 2004. Releases from 2004 on are beneficiaries of increased market penetration of multicore machines, increasing pressure on MySQL to use multithreading for performance.

Goals We will attempt to answer the following questions using behavioral information: (1) Has synchronization in MySQL changed through versions? (2) Has the amount of time in critical sections changed? We will use these questions to judge if MySQL developers have improved at multicore development since the widespread availability of multicore systems.

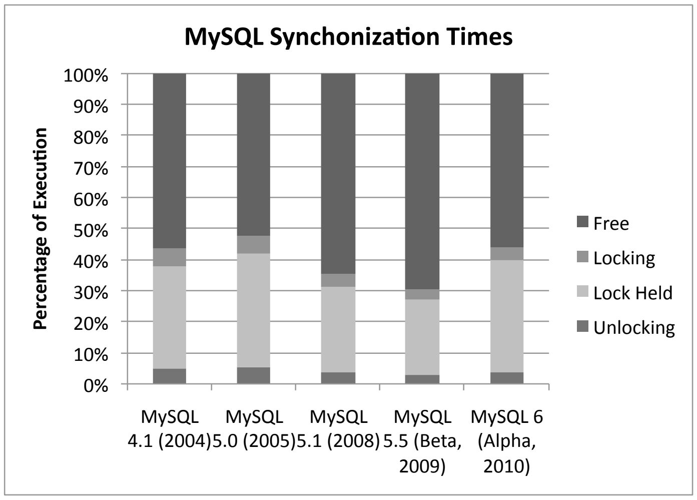
Necessity of LiMiT As in case study A, we are examining fine-grained program sections: lock acquires/releases and critical sections. To avoid perturbation, interference from multiple threads and error introduced by sampling, we require **LiMiT**'s low-overhead reads, process isolation and precision. Sampling is a poor option for the same reasons as given in case study A.

Experimental Setup To answer these questions, we intercept mysqld calls to the pthread library's locking routines to insert timing instrumentation. All versions of MySQL were compiled and executed on identical systems, so they all use the same, recent version of pthreads. As input, we run the "sql-bench" benchmark suite supplied with MySQL.

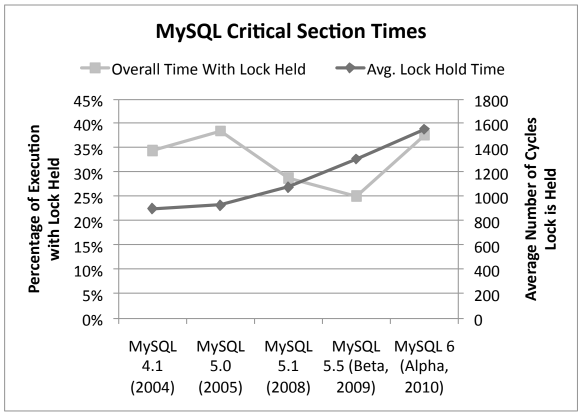
Results The results of this study are shown in Figure 3.10. They indicate that synchronization efficiency has increased since the 4.1 series, first introduced in 2004. Figure 3.10a examines overall times in synchronization and critical sections. Figure 3.10b rehashes the critical section results from the previous chart and overlays the average lock held time. Finally, Figure 3.10c examines the number of static and dynamic locks observed during execution. There are several interesting points to note:

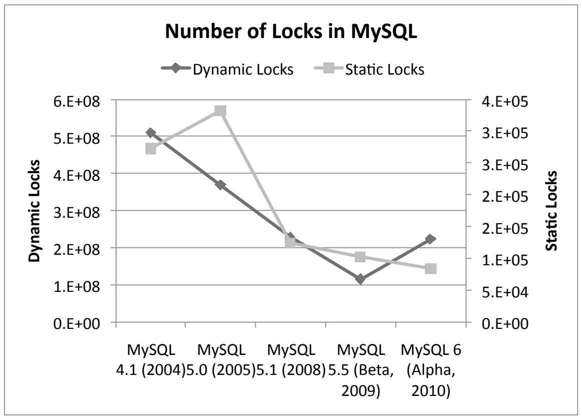
Average Lock Held Times MySQL developers have decreased the total amount of time spent with locks held while simultaneously increasing the average amount of time each lock is held. This implies that the functionality of multiple critical sections has been combined. For low-contention critical sections, this increases overall efficiency by avoiding lock overheads.

Lock Granularity The number of static and dynamic locks have both decreased. This implies that – on average – lock granularity has increased. Although this could increase



(a) Locking and Lock Held Times





(b) Lock Held Times

(c) Static and Dynamic Locks

Figure 3.10: A history of synchronization in MySQL. With the exception of MySQL 6 (a likely un-optimized alpha-quality version), time with locks held and time getting locks (contention and overhead) has decreased since version 4.1.

contention, it has not come at that cost, so this granularity shift has likely been carefully tuned.

Alpha Version MySQL 6, the alpha version, is an outlier with respect to recent versions. This is likely because it has not yet been optimized with respect to locking and new features have been implemented in overly conservative fashions.

To answer our initial questions, both synchronization overheads and critical section times have decreased over time. These performance improvements clearly show that developers have become more skilled, likely a result of multicore availability as parallel machines were not commonly available to hobbyist hackers before 2004.

Implication for Architects (#7): Performance Counter Utility While this is primarily a software engineering/project management study – and to the best of our knowledge the first study to use precise performance counters for software engineering – there is a very important take away point here for computer architects: there is a potentially broader consumer base for on-chip performance counter data beyond computer architects, OS and compiler writers. Computer architects should take this into consideration when designing future hardware monitoring systems. Broadly, this means that monitors should be optimized not to capture just the common execution cases but also uncommon cases which are interest in domains such as software engineering and security.

3.7 Related Work: Performance Counter Studies

Using performance counters to study workloads is common. Many studies have used them to point out deficiencies in benchmarks or microarchitectures. In a classic paper, Emer and Clark [49] constructed hardware counters which allowed them to construct histograms of microcode execution on the VAX-11/780. Phansalkar *et al.* [129] used performance counters to study redundancy in SPEC CPU2006 [67]. Ailamaki *et al.* [2] characterize database systems with performance counters. These are but several examples of characterization studies using performance counters.

Benchmark suites are also often studied using performance counters. BioBench [3], is developed and then analyzed with performance counters. Jaleel *et al.* [79] study the last level cache behavior bioinformatics workloads. Ferdman *et al.* [55] use performance counters to examine the microarchitectural characteristics of modern cloud workloads like web servers and MapReduce. They conclude that existing benchmarks do not represent accurately represent these applications and that modern high-end processors are improperly provisioned for them, both similar to the conclusions that we have made.

3.8 Conclusion

This chapter makes the following contributions: (1) We have introduced a lightweight, precise interface to performance counters on contemporary hardware. (2) We have conducted detailed case studies to demonstrate the utility of precise monitoring to architects. (3) Based on data collected with **LiMiT**, we offer new insights on program behavior which were not possible with existing tools.

As a demonstration of the usefulness of precise performance monitoring capabilities offered by **LiMiT**, we conducted three case studies on current web workloads. These studies lead us to the following conclusions:

• A new benchmark suite is recommended for research in computer architectures for the cloud era because traditional multithreaded benchmarks have different execution characteristics than multithreaded applications frequently used today.

• Web applications tend to have many very short critical sections which could be sped up with architectural support for lighter weight synchronization. Since the total overhead of lock acquisition and release is about 13% and 8% for Firefox and MySQL respectively, speedups in that range may be possible.

• Dynamically linked libraries and kernel code suffer from poor microarchitectural performance and also make up substantial portions of run time for system applications. Further research to enhance this performance could significantly accelerate web workloads.

• Performance counters have far wider applicability than just computer architecture (*e.g.,* software engineering) and architects designing performance counter systems should consider other applications.

These insights were made possible by precise, low-overhead performance monitoring capabilities provided by the **LiMiT** tool. These features allow monitoring of parallel programs more precisely than existing sampling based tools. In **LiMiT** we revisited and re-architected existing performance counter access methodologies (which had not been revised in the past decade). Specifically, we used novel kernel/user space cooperative techniques to allow user space readouts of performance counters. As a result, **LiMiT** is at least an order of magnitude faster than its existing state-of-the-art alternative, and reduces instrumented execution overheads significantly. In short, **LiMiT** can read virtualized counters in less than 12 nanoseconds, allowing precise measurements at finer granularities than have ever been studied.

Chapter 4

Measuring Side-Channel Vulnerability

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 in microarchitecture. 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 chapter, 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. We also conduct a detailed case study of on-chip memory systems. In it, 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.

4.1 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 4.1 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. 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 an entire 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 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 [126]. These accesses

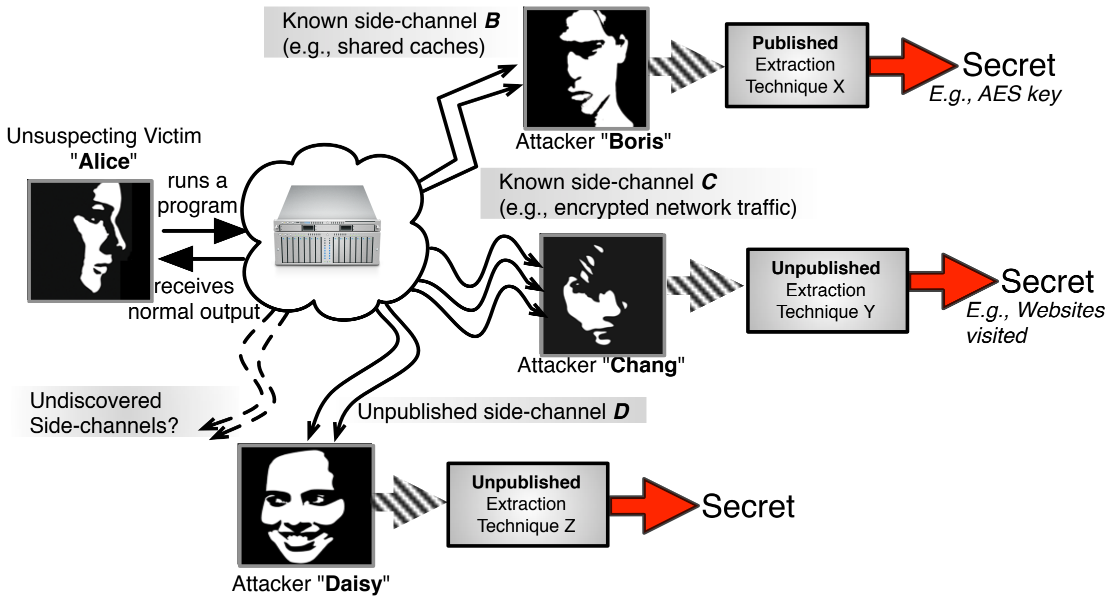
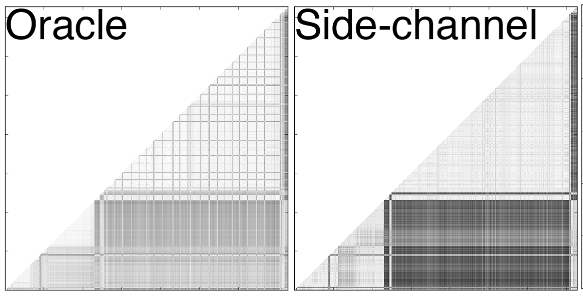

Known side-channel B
(e.g., shared caches)
Attacker "Boris"
Published Extraction Technique X
Secret
E.g., AES key
Unsuspecting Victim "Alice"
runs a program
receives normal output
Known side-channel C
(e.g., encrypted network traffic)
Attacker "Chang"
Unpublished Extraction Technique Y
Secret
E.g., Websites visited
Undiscovered Side-channels?
Unpublished side-channel D
Attacker "Daisy"
Unpublished Extraction Technique Z
Secret

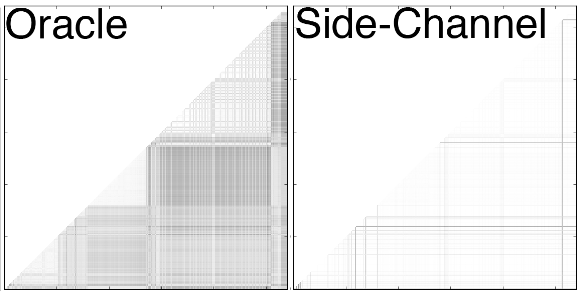
Figure 4.1: Information leaks occur as a result of normal program execution. Alice's actions can result in side effects. Attackers can measure these side effects as "side-channel" information and use it to extract secrets using known or unpublished attack techniques.

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 [113].

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 [70, 134] is well known and used in computer architecture. In light of this observation, side-channel attackers appear to 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 [63, 126]. 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 [29]. *Given this commonality of side-channel attacks, our key*



(a) A high SVF system (≈0.77)



(b) A low SVF system (≈0.098)

Figure 4.2: Visualization of execution patterns using similarity matrices. Each triangular matix compares a point in time with every other point in time within an execution trace. 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.

insight is that side-channel information leakage can be characterized entirely by recognition of patterns through the channel.

Figure 4.2 shows an example of pattern leakage through two microarchitectures, one of which transmits patterns readily and one of which does not. In Figure 4.2a, we can observe many of the visual patterns in the oracle matrix (on the left) can also be observed in the side-channel matrix (on the right). This intuitively demonstrates a leaky system: if the data which generated these patterns in the oracle is sensitive, then the side-channel attack has gained a good deal of information about those data. In Figure 4.2b, however, we observe that only some of the visual features from the oracle matrix can be seen in the side-channel matrix, and not all appear accurate. This visual example also nicely corresponds to the SVFs of the systems from which these data were collected: the visually leaky system has a high SVF, indicating high leakage, whereas the second system has a relatively low SVF.

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

	Example Insecure CPU	Example Secure CPU	Similar to attacked CPUs in [126], [63]	
SVF	0.86	0.01	0.73	0.27
SMT	2-way	1-way	2-way	2-way
Cache Sharing	L1	L2	L1	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 4.1: 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 [63, 126]. 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.

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 4.1. While the attack [126] on the 0.73 SVF system was relatively simple, the less vulnerable 0.27 system's attack [63] 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 [63, 121, 122, 126]. 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.

The primary contributions of this chapter are: (1) 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 explored before. (2) We 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.

4.2 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.

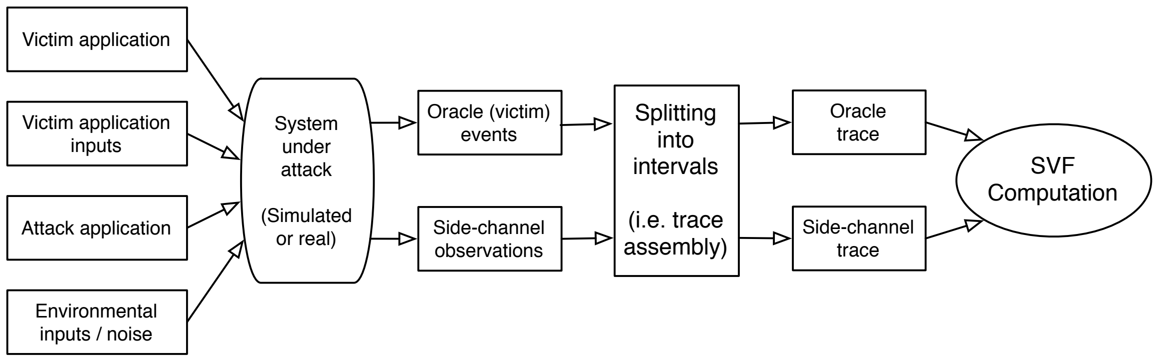

Victim application
Victim application inputs
Attack application
Environmental inputs / noise
System under attack (Simulated or real)
Oracle (victim) events
Side-channel observations
Splitting into intervals (i.e. trace assembly)
Oracle trace
Side-channel trace
SVF Computation

Figure 4.3: The SVF data collection and analysis process

4.2.1 SVF Context & Applications

SVF is an experimental measurement framework, meaning that rather than measuring a system's leakage in a vacuum, it measures leakage for a particular execution of a particular victim application with particular inputs while running a particular attack application. A overview of this method is shown in Figure 4.3.

Measuring SVF To measure SVF, victim and attack applications are executed on the test system, which may be simulated or real (if it is possible to accurately collect runtime data on the real system). During execution, events of interest are monitored and recorded. These events include data which the victim would like to keep secret – possibly bits in an encryption key or memory accesses. The events also include observations (measurements) which the attacker is able to make. After execution, the events are assembled into a pair of traces (oracle and side-channel) by splitting the events up into time intervals to form time series traces. Finally, SVF is used to compute the leakage between the two traces, as defined in Section 4.2.3.

Computation Dependencies As an experimental measurement, any SVF measurement will depend on a variety of factors, some of which are likely to vary dynamically (from run to run) and some of which may be statically defined for a set of experiments:

Victim Application & Inputs Since attackers are attempting to sense the data-dependent side effects of a victim's execution, both the victim and some input data must be modeled and can affect SVF. During execution, the victim is monitored for critical events.

Attack Application SVF measurement is based on correlation between victim execution and attacker observations. Therefore, an attack must be modeled and monitored to provide these observations.

Environment As in any experiment, environmental noise could affect the system being monitored. In a simulated setting, this noise is likely to be non-existent, though in a real system the effects of external interference could be significant.

System The system itself models or otherwise governs how the two applications execute and their interactions. We would expect the system to have the most drastic effect on leakage since leakage is – to a first order approximation – an artifact of resource sharing between the applications in the system. Systems with little or no sharing have little possibility for leakage, and systems with more sharing potentially leak significant amounts of data.

Events of Interest Another parameter which must be defined is what events are of interest and thus be monitored for later use to compute SVF. The choice of attacker events is reasonably obvious – it is the observations that the attacker has been able to make. The victim (or oracle) case is less obvious. Ideally, the events are secret data as they are being used; however, these secret data can be difficult to define, especially without making them application specific. For instance, bits in an RSA encryption key may be most informative for leakage in OpenSSL, but don't apply to other applications. Alternatively, memory addresses have been indirectly used in side-channel attacks, so memory loads and stores (and the addresses they touch) could be a useful application independent event stream.

Interval Splitting SVF computation operates on two discrete time series traces which are of equal length and aligned in time. To accommodate this requirement, we break up the events into intervals and combine all the events for each interval into a single data item. For instance, all of an attacker's observations within an interval may be combined into a single vector.

SVF Computation Specifics Finally, there are a number of parameters within the SVF computation which can be customized. For example, SVF needs to be able to compare the data items in each time step to items in other time steps via a distance function. This function can be defined in any number of ways. The SVF computing specifics are discussed formally in Section 4.2.3.

Dependence on Victim and Attacker Applications Ideally, we would able to measure the leakiness of a system independent of as many factors as possible. We suspect, however, that measuring leakiness independent of the victim and attack applications may be impossible for one basic reason: the interaction between the attacker and the victim may affect leakage. This could occur in two basic ways:

1. The method by which an attacker probes a system is critical and potentially affected by a variety of system factors, including the victim. For example, for a cache side-channel attack application may continuously scan a cache shared between itself and the victim. It seems logical that the speed at which this scan occurs could affect the quality of information obtained by the attacker – faster scans translate to more information per second. Further, this cache scanning speed could be affected by the victim itself – if the victim has many memory operations, it will use more resources in the cache, slowing down the attacker.

2. The execution of the attacker is likely to affect the execution of the victim. If, for example, the attacker operates by probing a shared resource, it is likely to slow down execution of of the victim. This slowdown could increase the effective leakiness of the system since it may make critical events and side effects easier to observe.

Applications As an experimental metric like performance or power, SVF can be used for a variety of different applications. For instance, one could vary any of the parameters listed above to determine its affect on leakage. Later in Section 4.5 we describe a case study wherein the system executing the applications was varied in a large design space. This allowed us to determine the effects of various processor design parameters on leakage. However, this is only one application. It may also be interesting to try a series of different

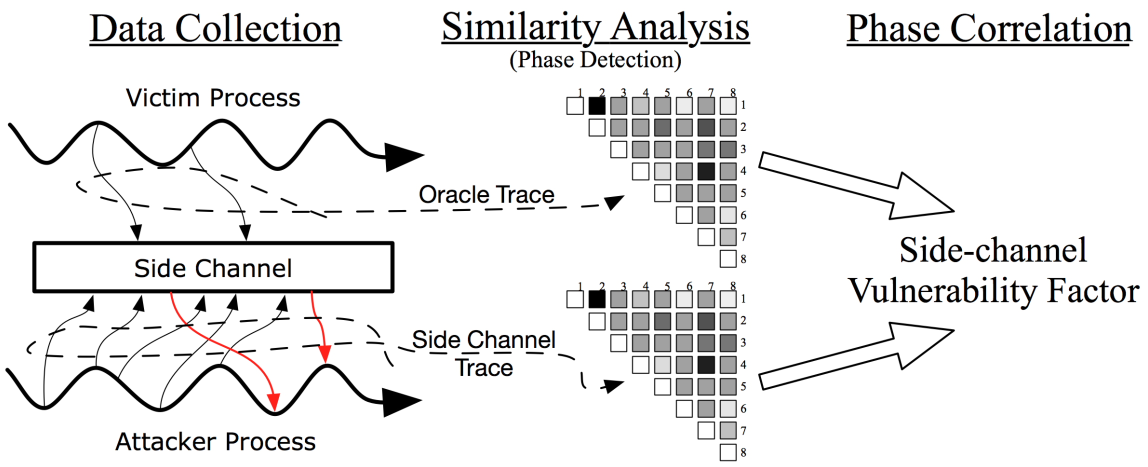


Figure 4.4: An overview of computing SVF: two traces are collected, then analyzed independently for patterns via similarity analysis, and finally correlation between the similarity matrices is computed.

attack applications which probe the system differently to determine more effective methods of attack. Similarly, software engineers could execute different versions of their software to determine if software can be made to leak less. Any number of different experiments are possible.

4.2.2 SVF Computation Overview

An overview of SVF computation is shown in Figure 4.4. 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.

4.2.2.1 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 SVF detects 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.

4.2.2.2 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.

4.2.2.3 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 4.2a). 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 4.2b. 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 [125], 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.

4.2.3 SVF Definition

Now that we have qualitatively described SVF's various parts, this subsection defines each component and SVF overall. Table 4.2 defines many of the terms and symbols used in this section.

Traces We define both the oracle and side-channel traces as finite sequences of symbols from two different alphabets. The length of both traces must be equal. The alphabets, however, can be nearly anything. Symbols in the oracle alphabet, for instance, could represent bits in an RSA encryption key or memory addresses accessed. The side-channel alphabet could represent cache hits/misses or even real numbers representing power usage.

Distance Functions & Similarity Matrices Since each trace has its own alphabet, they are not directly comparable. Instead, we require that for each alphabet, one define a distance function which can compare any two symbols. This function will be used to detect phases in each trace using similarity matrices, as defined in Table 4.2.

SVF SVF is defined as the correlation between M_O and M_S, two triangular matrices of the same size. We first build two sequences X and Y both of length $|T_O|^2 - |T_O|$ containing the reals from M_O and M_S. We then compute the Pearson Correlation Coefficient (PCC) between X and Y. PCC is defined [125] as:

$$\frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{4.2}$$

Term	Symbol	Definition
Oracle Trace	T_O	A finite sequence of symbols from an alphabet A_O
Side-channel Trace	T_S	A finite sequence of symbols from an alphabet A_S
Oracle Distance Function	D_O	A function which computes the distance between two symbols from A_O. $(A_O, A_O) \mapsto \mathbb{R}$
Side-channel Distance Function	D_S	A function which computes the distance between two symbols from A_S. $(A_S, A_S) \mapsto \mathbb{R}$
Similarity Matrix	$SM_{(T,D)}$	A triangular matrix of size $\|T\|x\|T\|$ computed from trace T using distance function D. Each matrix entry is defined as: $SM_{(T,D)}(i,j) = \begin{cases} D(T(i),T(j)), & \text{if } i > j \\ \text{undefined}, & \text{otherwise} \end{cases}$ (4.1)
Oracle Similarity Matrix	M_O	$SM_{(T_O,D_O)}$, the resulting similarity matrix given an oracle trace and distance function.
Side-channel Similarity Matrix	M_S	$SM_{(T_S,D_S)}$, the resulting similarity matrix given an oracle trace and distance function.
Side-channel Vulnerability Factor	SVF	Correlation between M_O and M_S. Any correlation metric could be used. In this chapter, we use Pearson Correlation Coefficient [125].

Table 4.2: Definitions of terms to compute SVF

4.2.4 Optimization: Partial Matrix Correlation

There is a practical difficulty in computing the SVF as we have defined it above. While most of the steps' runtimes are linear with the size of their inputs, computing the similarity matrix exactly is in $\theta((\frac{n}{c})^2)$ for n as the program run time and c as the interval size. As a result, this computation does not scale well to long running applications with short intervals.

Fortunately, it turns out that we can instead compute an approximation of the full matrix correlation. This is possible because similarity matrices are typically very redundant, seen visually in Figure 4.2a. The reason for redundancy follows from the triangle inequality property of distance computations. Specifically, given intervals A, B and C, if A is similar to B and B is similar to C then A is also similar to C. In the matrix, however, all three similarities are reflected. Additionally, recall that we use n intervals to create an n^2 matrix yet no new information is created in the conversion (it is analysis only) though much more space is used, so from an information-theoretic stand point, the similarity matrix is extremely redundant.

As a result of this redundancy, we have found that we can compute a subset of the matrix and find the correlation for only that subset, as long as the same subset is computed for both the oracle and side channel matrix. In practice, we have found that randomly selecting a subset proportional to the number of intervals yields an accurate approximation. Further, this keeps the entire SVF computation complexity linear with the number of intervals (and thus victim application running time).

4.3 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.

4.3.1 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.

4.3.2 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. Creating an absolute metric completely independent of an attack would require a measurement of the total amount of information leaked during a computation. It is not known 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 architects do when reporting energy efficiency.

Experimental Methodology One pitfall of SVF is that it cannot analyze a particular system in a vacuum. Rather, it is computed for particular executions so is subject into input bias; the application being run, environmental noise, and a variety of factors affect SVF. As a result, SVF will likely have to be run on a benchmark set rather than single application. This creates the need to select an interesting benchmark set, both in terms of programs and oracle/side-channel trace selection.

SVF Validation We have presented intuitive case that SVF is a good descriptive metric which should be used to evaluate side-channel leakage. We have not, however, provided an quantitative evaluation showing that SVF is effective for this task. The reason for this is simply that such a study likely cannot be done. Such a study would have to show that in high SVF systems, leaks are both easily exploited to gain accurate data. This is relatively easy to show by executing well known attacks on high SVF systems. However, the contrapositive (that low SVF systems are either difficult to exploit or only inaccurate information can be gained) is far more important to show. If we cannot show this, then the study would not be able to indicate that lowering a system's SVF leads to increased security, the primary application for SVF. Unfortunately, it is probably not possible to quantitatively and convincingly show that low SVF systems are resilient to attack (and at the very least would require a breakthrough in security analysis) as at requires a provably optimal attacker. Indeed, were such analysis possible, we would simply use this method in place of SVF! As such, we consider SVF to be a strong descriptive metric but – like nearly all practical security methodologies – not provably infallible.

4.4 Case Study: Memory Side-Channels

Since shared cache microarchitectures have been exploited in the past [63, 121, 122, 126], 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.

4.4.1 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 4.5 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 re-measuring 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. Of course this is an idealized model for the attacker; various system effects distort the measurements, making the side-channel noisy.

In the example of Figure 4.5, 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

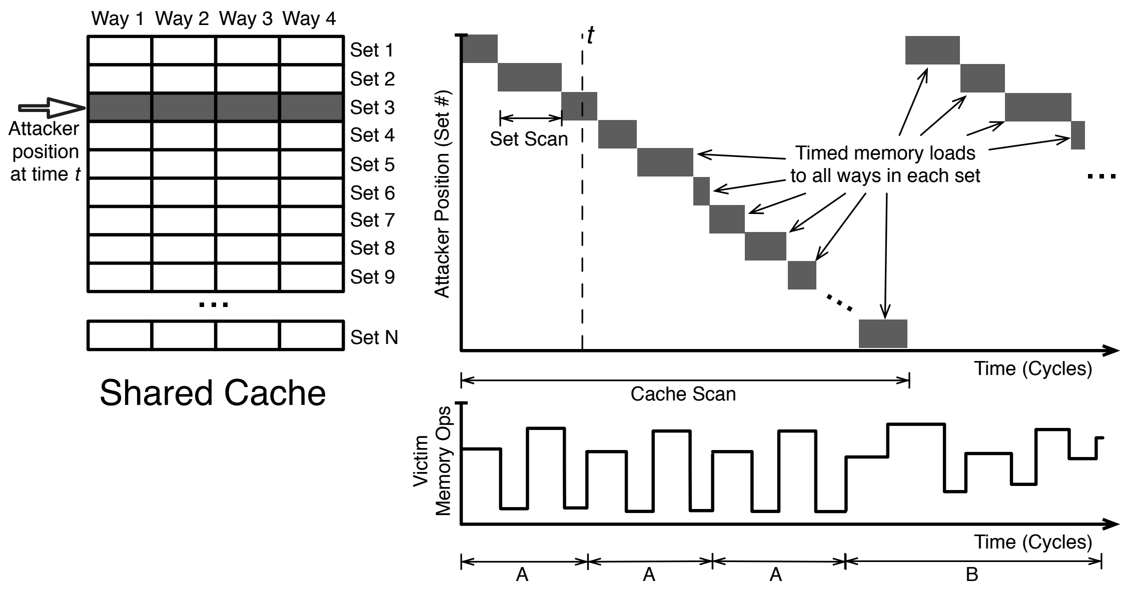


Figure 4.5: In cache side-channel, an attacker times loads to each set in a shared cache one set at a time. The amount of time to complete the loads indicates the amount of contention in the set, thus leaking data about a victim's working set.

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 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.

4.4.2 SVF Computation Specifications

In this section we illustrate how SVF can be computed and used to evaluate modern on-chip memory system. The aim of this study is to understand how microarchitectural aspects of the memory system such as cache configurations, prefetchers, etc., affect SVF. A meta-goal of this section is to outline a template/methodology for designers interested in measuring SVF.

As a first step for this study we will define our experimental world using of three key components: the victim program of interest, attacker capabilities, and assumptions about the microprocessor. We will then describe the memory system aspects we are interested in studying and explain why we consider them to be interesting for vulnerability analysis. Following this we will describe the type of analysis we use to compute SVF; results are presented in the following section. The apparatus for this study is a simulator that is setup to execute a side channel attacker alongside a victim application in an environment where they share microarchitectural structures in the memory system and the core pipeline.

Side-channel As demonstrated in Figure 4.5, 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 4.2.2.2 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.

4.4.3 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 avoiding 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.

4.4.4 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 implements 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, in-

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 4.3: Fixed cache design parameters

creasing the SVF of a particular system. Thus our simulation results should be somewhat conservative.

4.4.5 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 4.5, 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 [155]. PRS maintain a permutation of the sets in the cache, providing a time-varying hashing function. 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.[1] 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 3%. Although our implementation and algorithm for set permutation is different from Wang and Lee [155] we obtain similar results; in particular, we find that PRS can improve security.

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 [59], GHB/PCCS, and GHB/PCDC [119]. 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

[1]It's important to note that our usage of PRS is different from RPCache.

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, however, 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 4.3.

4.5 Case Study 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.

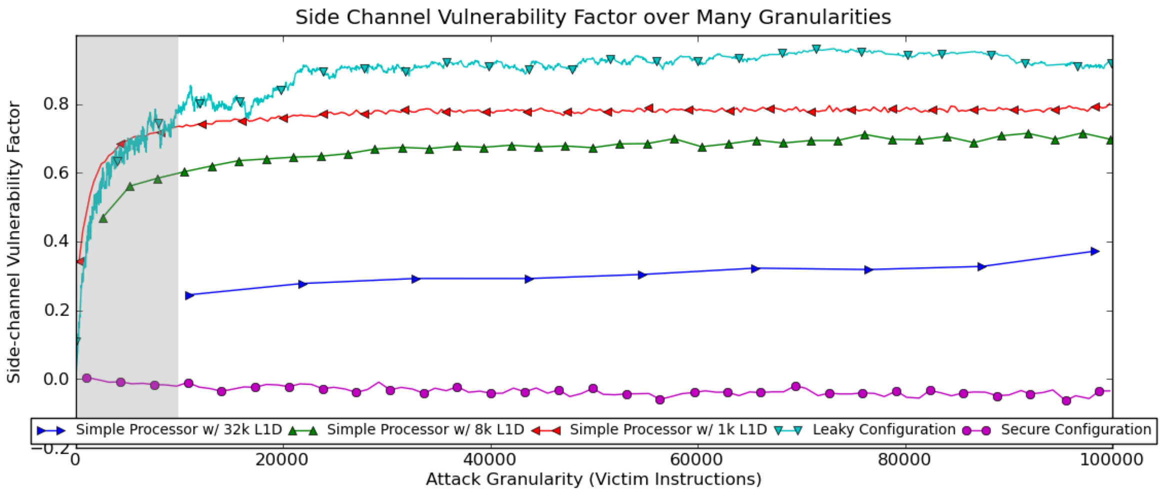
4.5.1 Interval Sizing

As discussed in the last section and Figure 4.5, 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 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 them beginning with the finest possible, the time it takes for the attacker to complete one scan of the all the cache lines. Figure 4.6 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



	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 4.6: 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.

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 chapter, due to space considerations, we present only a subset of the intervals sizes shown in Figure 4.6. 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 4.6. 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 4.7 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.

4.5.2 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 4.7 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 config-

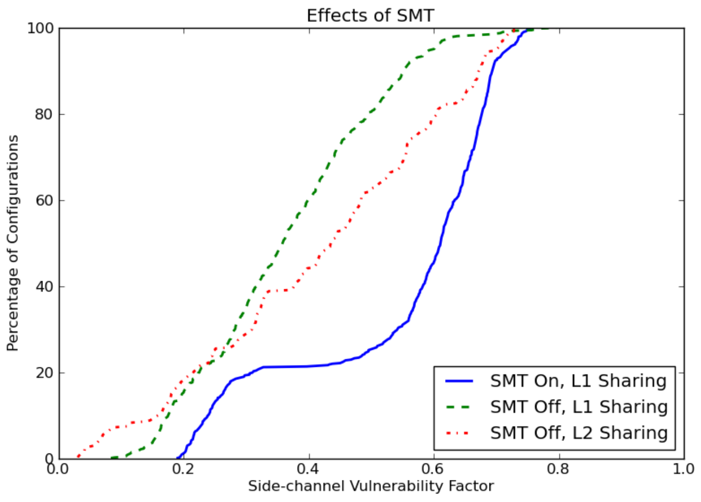


Figure 4.7: This cumulative histogram shows the percentage of configurations with various sharing configurations which have SVFs no more than the value on the X axis. For instance, all SMT configurations have SVFs of at least 0.2 and only 40% of them have SFVs less than 0.6. Without SMT, however, the 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.

uration 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.

4.5.3 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. Figures 4.8, 4.9, and Table 4.4 show 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.

Cache Size One of the largest determinants of SVF is cache size. Consistent with the data in Figure 4.6, the cache size graph in Figure 4.8 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

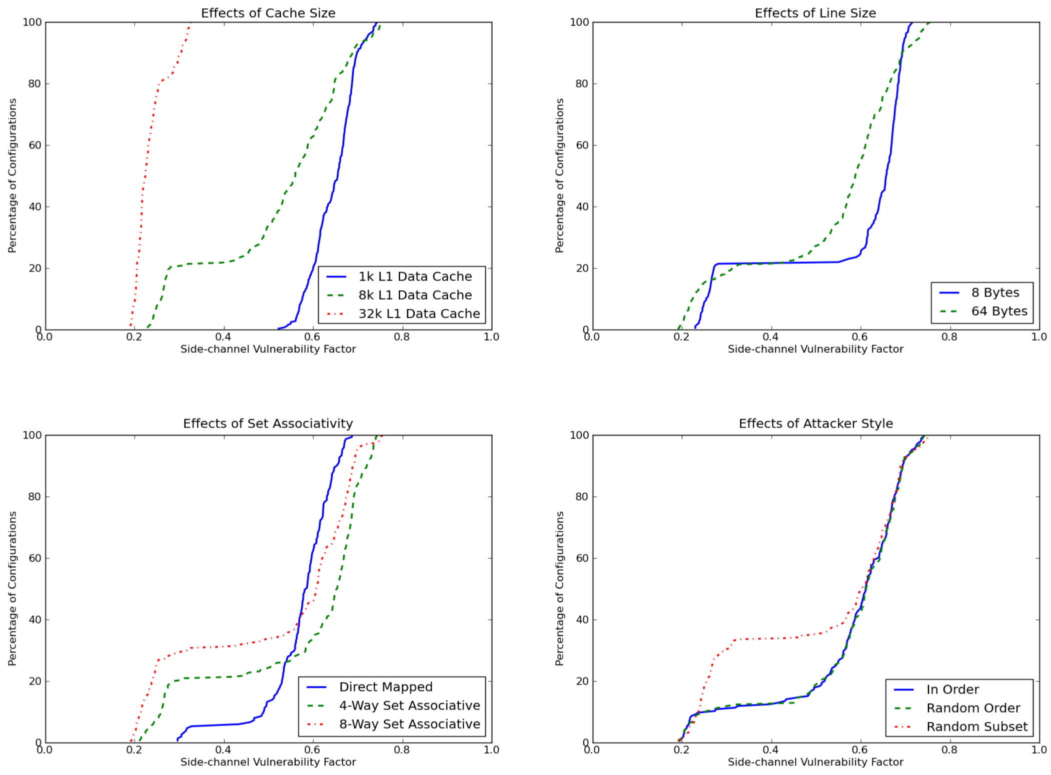


Figure 4.8: Cumulative histograms demonstrating how cache size, line size, set associativity and attacker style effect SVF in systems with SMT and a shared L1 and no protection (like partitioning and random eviction). In each graph, a set of features are selected and we draw a cumulative histogram with 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.

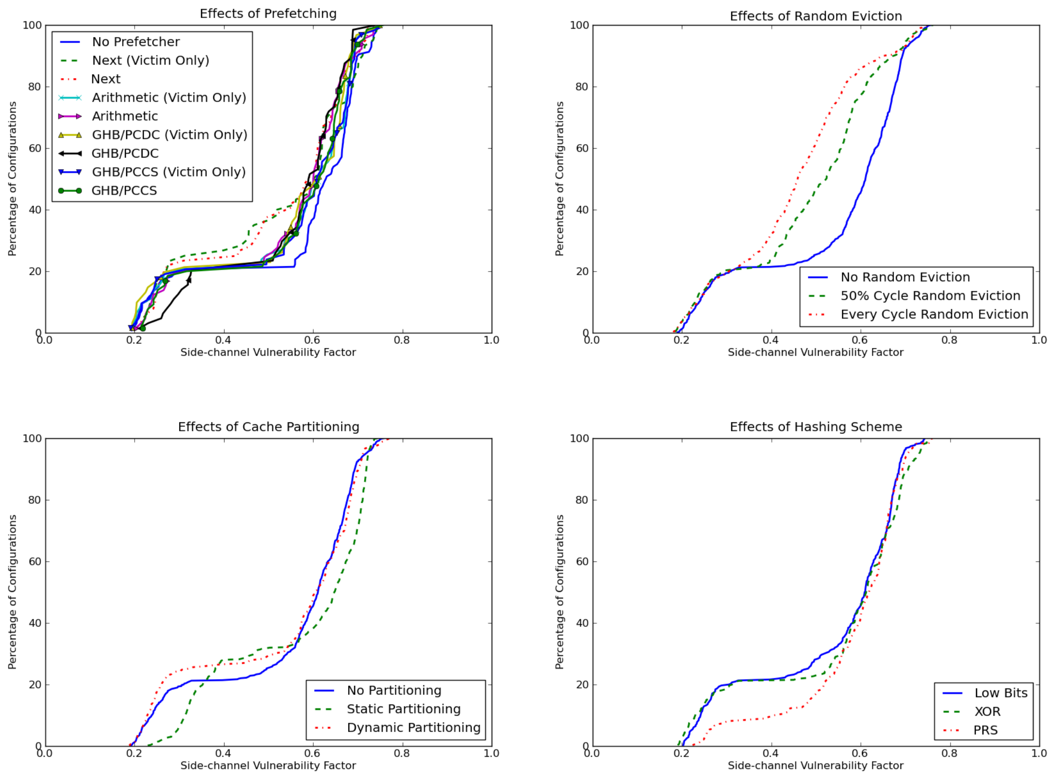


Figure 4.9: Cumulative histograms demonstrating how prefetching, random eviction, cache partitioning and hashing scheme effect SVF in systems with SMT and a shared L1 and no protection (except for the last two).

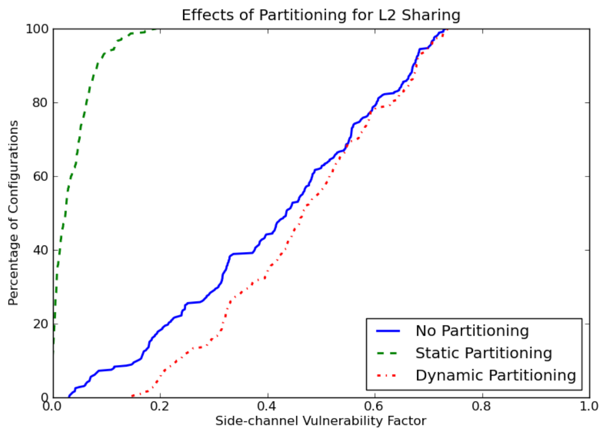


Figure 4.10: Static partitioning is extremely effective when SMT is turned off. Since this disables pipeline side channels and static partitioning disables cache *content* side channels the only remaining side channels are shared cache buffers and ports, which are not terribly effective side channels. Additionally, we see that a simple dynamic partitioning mechanism can *itself* leak information.

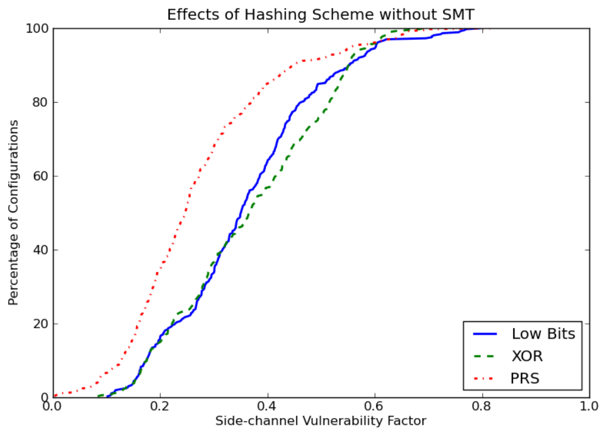


Figure 4.11: In some SMT configurations, our implementation of permutation register sets (PRS) can leak information. However, this is largely because it does not address pipeline side channels, yet slows down the victim. If we turn off SMT yet still share the L1 cache (as in this figure), we see that PRS obscures the side channel as expected.

Configuration Option		# of SVFs	SVF Mean	SVF Median	SVF S.D.	SVF Min	SVF Max
*	*	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	1k	2,213	0.499	0.598	0.226	0.002	0.761
	8k	1,194	0.475	0.544	0.219	0.000	0.771
	32k	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	PRS100	1,330	0.490	0.558	0.207	0.004	0.761
	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	790	0.467	0.519	0.219	0.002	0.756
	Arithmetic	783	0.471	0.561	0.221	0.001	0.765
	GHB/PCDC	787	0.462	0.539	0.214	0.001	0.757
	GCB/PCCS	787	0.475	0.567	0.219	0.000	0.760
	None	786	0.487	0.588	0.235	0.000	0.771

Table 4.4: Summary of design space exploration showing the number of microarchitectural configurations which completed a cache scan in under 10k victim instructions. These data indication that some hardware features have a greater effect on 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.

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.

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.

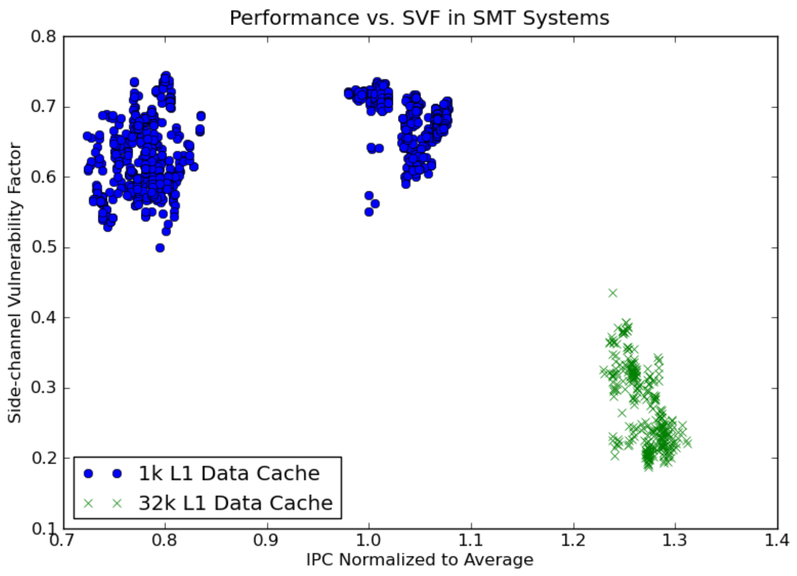
Random Eviction One might expect randomly evicting cache lines to introduce noise, and thus degrade the side-channel. Our simulations indicate that this is true, but only to a relatively small extent. Further, this technique is only effective on about 70% of cache configurations.

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 4.7, 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 4.10 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 other timing and pipeline channels are able to get more information. However, PRS is not always more leaky; in Figure 4.11 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. Also, using different cache set swapping algorithms could easily improve security. RPCache, a different usage of PRS [156] does exactly this.

4.5.4 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 4.12 we examine performance trade-offs 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.



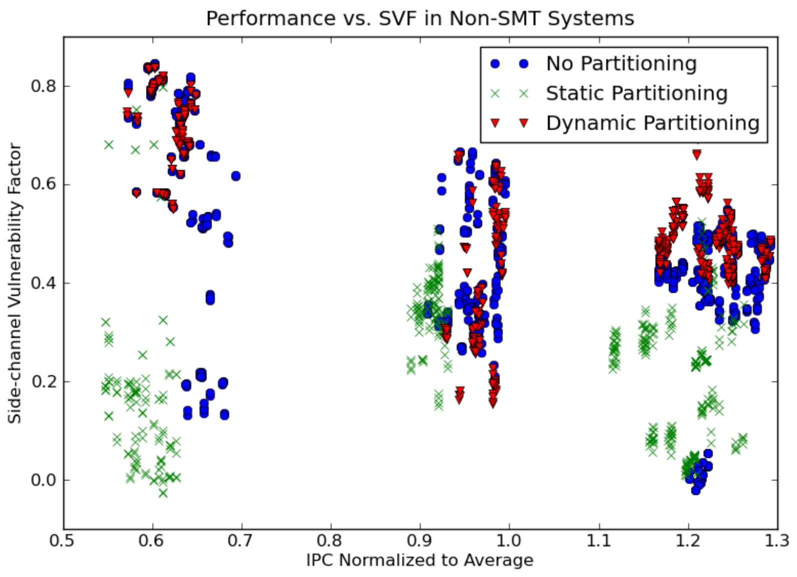


Figure 4.12: Many security proposals result in decreased performance. However, these results indicate that this need not always be the case. For instance, in SMT processors (top figure) increasing the cache size both decreases SVF and increases performance. In non-SMT systems (bottom figure) there exist high-performance systems with very low SVFs using both no cache partitioning and static partitioning.

4.5.5 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 chapter.

4.5.6 Protection Mechanisms

In addition to the two protection mechanisms reviewed earlier this section (random eviction and PRS), we evaluate two others. The first is a mechanism from our group called TimeWarp. The second is similar to PRS from the last section which uses a different algorithm for swapping cache sets. Both of these mechanisms were implemented in a different version of the simulator, thus these results are not comparable to the others so we present them here in a separate subsection.

TimeWarp TimeWarp [109] is a proposal to mitigate side-channels by obscuring an attacker's ability to measure time. In short, TimeWarp "fuzzes" the results of timing functions like reading the processor's timestamp counter and adds a delay to each timer read as a penalty. As a result, the attacker shouldn't be able to make the accurate timing measurements necessary for published attacks. We implemented TimeWarp in our SVF simulator

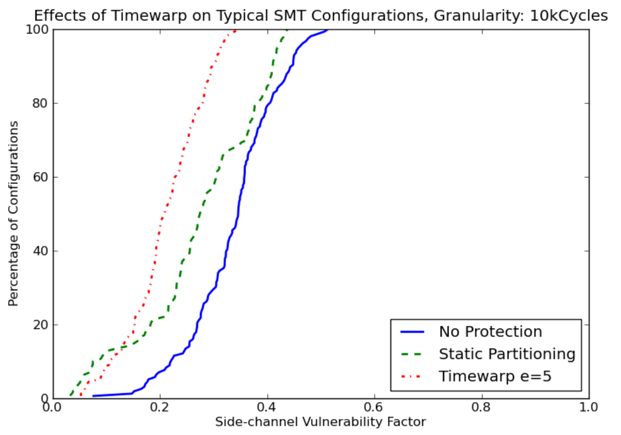


Figure 4.13: SVF measurements of Timewarp systems at the 10K victim instruction granularity. We see than a relatively small Warpfactor of 5 ($e = 5$ or up to 32 cycles) somewhat obscures the side channel. At $e = 13$, however, the attacker is not even able to complete a scan.

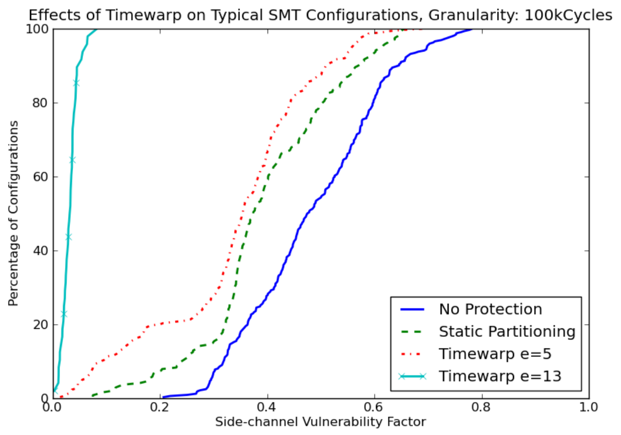


Figure 4.14: SVF measurements of Timewarp systems at the 100K victim instruction granularity. Again we see than $e = 5$ obscures the side channel. At this granularity the attacker is able to complete cache scans for the higher warp factor, so we see data for $e = 13$. Although some leakage still occurs, the side channel is heavily obscured.

and measured it for different levels of fuzzing (denoted by e, the TimeWarp factor). The results are shown in Figures 4.13 and 4.14 relative to statically partitioning the cache and no protection mechanisms. We see that for a fine granularity attack (10k instruction granularity), even a relatively small fuzzing factor begins to obscure the side-channel. A larger factor slows down the attacker enough that a single cache scan cannot complete in less than 10k victim instructions. In Figure 4.14, we increase the granularity of the attack to 100k victim instructions. Here, an attacker can complete a cache scan with a large TimeWarp factor, though we see that the side-channel is largely obscured.

RPCache RPCache [156] uses the same permutation register sets (PRS) as examined in the previous subsection in order to maintain separate logical to physical cache set mappings for the victim and attacker processes. However, in contrast to the simple policy we used in the last section (swapping two random cache sets every 100 loads), RPCache selects more intelligently. In this, whenever a process misses in a cache set, RPCache selects that cache set for swapping along with a randomly selected cache set. The inventors claim that this closes the cache side channel entirely. Figure 4.15 quantitatively evaluates RPCache using SVF in a system context. We see that RPCache's selection policy does indeed result in lower SVFs than our simple policy (PRS-100) on average. As we saw with statically partitioned caches, however, in the context of a system with SMT it does not make a large difference, likely because other side-channels (like a pipeline side-channel) also contribute heavily to leakage.

4.6 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

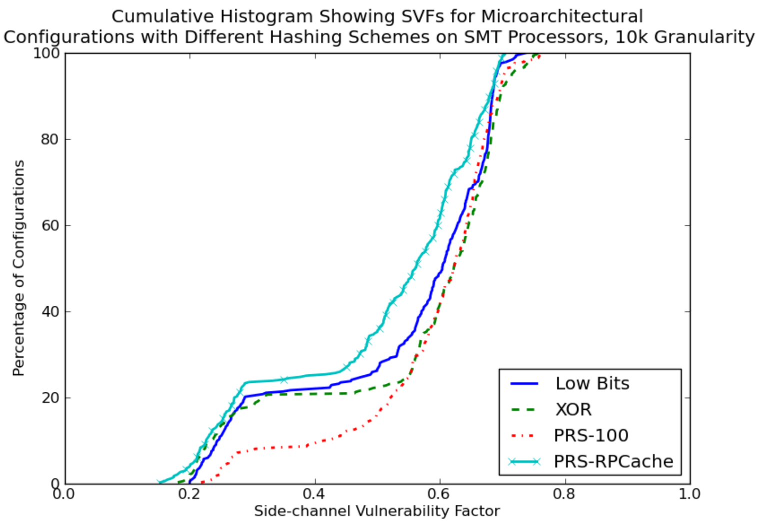


Figure 4.15: RPCache proposes to permute cache sets more intelligently. It seems to result in less leakage than our simplistic permutation algorithm in PRS-100, though it does not make a significant difference in our system context.

and countermeasures [89–91, 156, 157].

A classic (blue sky) goal in information flow security is non- intereference [133]. Strict non-interference ensures that an adversary can deduce nothing about secret inputs from public outputs. By definition, non-interference between two processes ensures that no side-channel leaks occur. For example, by physically isolating the computing resources of two processes, non-interference is assured. This goal has been difficult to achieve in a practical, efficient manner but there have been some recent investigation on this topic [149]. Given the difficultly of ensuring strict non-interference there has also been work to relax non-interference with quantitative estimates of how much information leaks, enabling leak/risk analysis. This area of study is called quantitative information flow and works in this area typically uses information theoretic measures and simple theoretical models. Extending this theory to complex software systems is being investigated [68]. SVF is the first practical experimental method to measure information leakage in hardware systems.

Traditionally, cryptographic systems have made the assumption that no resources are shared so there is no side-channel leakage. As a result, they assume the only information which can be seen by an attacker are messages exchanged between cryptographic systems.

Layer	Related Work
Algorithm	New cryptographic strategies [46, 85] are resilient to bounded amounts of leakage. They rely, however, on assuming a certain bound on leakage, requiring leakage characterization of the lower levels of the stack.
Software	A large body of work (reviewed by Sabelfield and Myers [133]) focuses on analyzing software via quantitative information flow. Backes *et al.* [11], for instance, use information flow to discover equivalence classes which map to secret data.
Operating System	No known work.
Architecture & Microarchitecture	Since cache side channels are well known, there has been some investigation in modeling/measuring cache leakage. Cache Side-channel Vulnerability (CSV) [162] attempts to measure cache leakage using an approach similar to SVF, but avoids using similarity matrices by limiting their scope certain types of caches. Dominitser *et al.* propose an analytical model to predict the amount of information leakage through cache side-channels [43]. The technique proposed in their paper tracks the fraction of the victim's critical items accessible in the cache to determine leakage.
Hardware Gates	GLIFT [149] adds information flow tracking (IFT) at the gate level to hardware designs. The IFT results can be used to detect interference between processes and thus verify non-interference or detect new leaks. It is possible that GLIFT could be extended to also characterize the amount of leakage.

Table 4.5: Efforts to model and/or measure leakage at various levels in the software/hardware stack.

Increasingly, however, it is recognized that it is impractical to avoid resource sharing, so models and measures of leakage are being created at various levels of the hardware/software stack: from the algorithmic layer to the gate level, as shown in Table 4.5.

SVF applies most directly to the architecture/microarchitecture layer. Our work differs from existing work 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.

4.7 Objections to SVF

Since the original publication of SVF in 2012 [40], there has been another publication from Zhang *et al.* [162] (which we will refer to as the "CSV paper") which voiced several objections to SVF and proposed an alternative metric called Cache Side-channel Vulnerability (CSV). The CSV paper contains a section entitled "Discussions on SVF" which contains three objections to SVF: Scope, Definition, and Measurements. Their arguments leading to these objections contain errors which we will correct here.

Scope Zhang *et al.* erroneously state that "the aim of SVF is to use a single metric to reveal the information leakage of the entire system for all side-channel attacks" and as a result, they claim that SVF's scope is overstated. SVF does indeed measure leakage of an entire system, however does so for a particular victim and attack. In fact, SVF is not a single metric – it is a way to define side-channel leakiness metrics.

Definition The critical piece of SVF is to compare the similarity matrices generated from two traces – a victim and attacker trace. The CSV paper takes exception to both the use of similarity matrices and the selection of traces used in the the cache side-channel case study we presented in this dissertation and in our original publication. This objection to the specific traces used in the SVF paper is somewhat superficial; their arguments may or may not have merit (depending on your perspective), but the SVF methodology allows any trace to be defined. More importantly, while Zhang *et al.* claim that the similarity matrices are unnecessary and propose eliminating them (using direct correlation instead), using them is in fact necessary in order to compare most types of traces. For instance, say one wished to compare a trace of bits in the victim's encryption key with cache misses which the attacker observed – how can correlation be directly calculated? The CSV paper implicitly assumes it can get around that problem by always correlating traces regarding cache sets to traces regarding cache sets, however this assumption only holds if one doesn't play with the cache's hashing policy, a oft-used trick to defend caches.

Measurement The CSV paper's final objection to SVF is that it does not separate system vulnerability from attacker capability. While this observation is accurate, the paper misses the fact that the system can directly affect attacker capability. To divorce the two, Zhang *et al.* propose to use a synchronous attack model, assuming that an attacker has the ability to frequently pause execution of the victim for a sufficient amount of time to probe it. Given that synchronous attacks exist [63], this could be a reasonable model, however it is actually a very strong attacker model.[2] In the more plausible asynchronous attack model – which the SVF paper (this chapter) used in its case study – the attacker's capability is directly affected by how quickly it is able probe shared resources – like scanning a shared cache in cache side-channel attacks. This speed can be directly affected by both the system under attack and the concurrently executing victim – *i.e.,* cache misses and pipeline resource contention will slow down the attacker (thus enabling the attack, actually). The point is that while Zhang *et al.* claim that we have not stated the attacker's capability, they miss that this capability is governed by the system and the method by which the attacker probes it, both of which are defined in the case study.

Intuitive Assertions Though not explicitly stated as an objection to SVF, one of the CSV paper's implicit objections to SVF is that the results of our case study do not match their intuitive expectations, which they claim to be "ground truths". In particular, they state the relative leakage of several cache configurations and claim that a good metric

[2]Were attackers allowed to arbitrarily interrupt execution of victims, denial of service attacks would be possible, so systems generally disallow this behavior. As a result, synchronous attacks must rely on other system vulnerabilities (such as the O/S scheduler).

should reflect these beliefs.[3] These "ground-truths" are intended to be self-evident and are backed up with some intuitive explanation. However, the authors again miss one of the major conclusions from the SVF paper: that intuition is difficult to apply in a system context. Side-channel attacks operate on a system as a whole, not just one small part of it. Further, complex systems are very difficult to reason about – their components' interactions can have odd side-effects, which are difficult to predict; there is a reason we have to run detailed simulations instead of using analytical models. As a result, **when defining a metric, intuition needs to be applied very carefully, if at all.**

For example, Zhang *et al.* claim that a statically partitioned cache should leak less than an unpartitioned cache. An intuitive assertion, for sure. Indeed, if the entire system were composed of *only* cache sets, we would agree with this assertion. However, the measurements which attackers make are affected by more than just contention for cache lines. The attacker's data are affected by contention for many resources: interconnect bandwidth, memory bandwidth, pipeline functional units, and MSHRs, just to name a few. As a result, the basic model of the system which motivates all of Zhang *et al.*'s assumptions – the model of cache conflicts as the only source of slowdown which the attacker observed – is insufficient for judging a full-system property like side-channel leakage.

4.8 Conclusion

In this chapter 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

[3]In fact, one of their assertions is that PRS and RPCache [155, 156] – work previously published by the same group – reduce leakage. It seems biased for the inventors of a scheme to propose it as a benchmark for the metric by which that very scheme should be judged.

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.

Part II

Analysis

Chapter 5

Finding Common Code Patterns

An important aspect of system optimization research is the discovery of program traits or behaviors. In this chapter, we present an automated method of program characterization which is able to examine and cluster program graphs, *i.e.,* dynamic data graphs or control flow graphs. Our novel approximate graph clustering technology allows users to find groups of program fragments which contain similar code idioms or patterns in data reuse, control flow, and context. Patterns of this nature have several potential applications including development of new static or dynamic optimizations to be implemented in software or in hardware.

For the SPEC CPU 2006 suite of benchmarks, our results show that approximate graph clustering is effective at grouping functions which react similarly to compiler optimizations. Graph based clustering also produces clusters that are more homogeneous (with respect to performance reaction to compiler optimization) than previously proposed non-graph based clustering methods. Further qualitative analysis of the clustered functions shows that our approach is also able to identify some frequent, though unexploited program behaviors. These results suggest that our approximate graph clustering methods could be very useful for qualitative program characterization.

5.1 Introduction

Identifying interesting program execution behaviors is important for creating optimized, secure systems. Today, program characterization is a laborious and increasingly time consuming process due to a combination of factors including the growth in number of applications, the wide variety of platforms these applications run on, and difficulty in the characterization process. To see some of the challenges in program characterization consider the case of SPEC benchmarks. Figure 5.1 plots the number of functions responsible for a certain fraction of the execution time. For example, five functions from *each* SPEC INT benchmark (of which there are 11), contribute to, on average, 67% of execution time. If SPEC FP is examined as well, the number of functions grows from 55 to 135. In fact, examining all of the functions in SPEC responsible for any non-trivial amount of execution time (at least 1%) requires characterizing about 300 functions comprising about 14,000 lines of code. This is a significant amount of code but is still small in comparison to real-life codes. The last data set shown in figure 5.1 is coverage data from the V8 Javascript Engine, a production library used in the Chrome web browser. The V8 profiling data indicate that the amount of code that must be characterized is very large – nearly 30% of its functions must be examined to cover 90% of its execution. Given this immense scale, automated program characterization could be much more comprehensive than manual characterization, potentially yielding more and more accurate qualitative insights into code behaviors.

To ameliorate these challenges researchers have proposed a wide variety of technologies including sophisticated code profiling, fast simulation techniques, and machine learning with performance counter data [38, 44, 118]. Some researchers have even proposed crowdsourcing approaches [60, 61]. In this chapter we propose a new technique to study program behavior which allows automatic identification of unique code behaviors across code bases by *approximately clustering program graphs.* By clustering similar graphs, we expose similar control and dataflow patterns in software, allowing one representative sample from each cluster to be studied rather than all graphs.

While we are not the first to observe the benefit of clustering [83], prior approaches have focused on clustering of non-graphical formats such instruction frequency, or microarchitecture dependent features such as cache misses or IPC measured from performance counters.

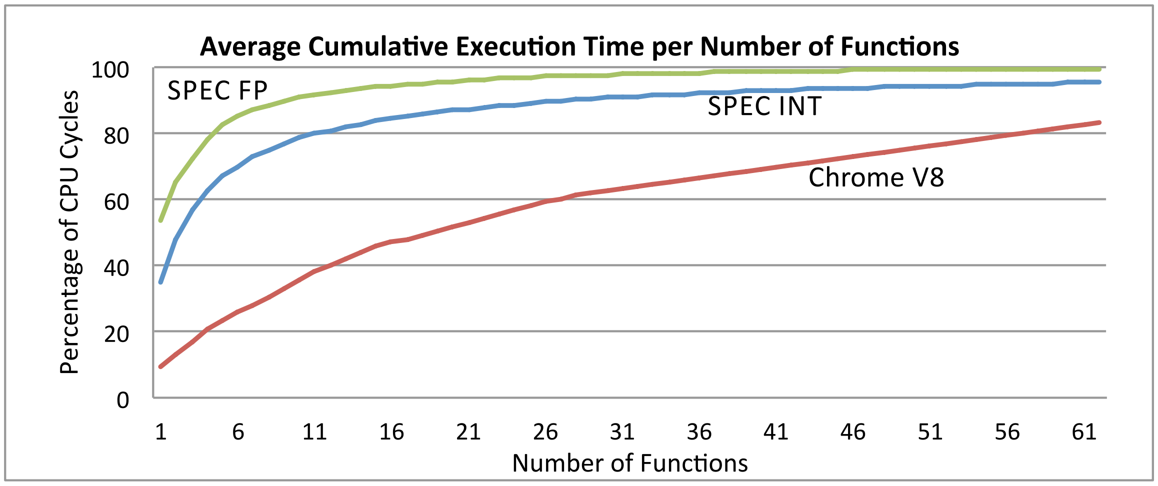


Figure 5.1: This chart plots cumulative program execution time (on the Y-axis) along with number of functions contributing to the execution time (on the X-axis). Measurements on 11 SPEC INT benchmarks and 16 SPEC FP (both averaged in the figure) and the V8 Javascript engine distributions all demonstrate that *many* functions contribute to total execution time significantly. This spread presents a significant challenge for program characterization.

Our technique is the first to propose clustering on program graphs and thus enables microarchitecture independent characterization of programs. Futher, graphical intermediate representations are a semantic step closer to algorithmic description and thus may offer more fundamental insights into program traits and lead to more comprehensive program characterization.

We adapt a decade old advance in identifying similarity in graphs [112] to create approximate graph clustering for program characterization. Traditional graph similarity methods such as isomorphism can determine if two graphs match, but are not useful for clustering because even very similar programs can produce slightly different graphs. Approximate graph clustering, on the other hand, instead of providing discrete answers, produces a continuous measure for similarity based on the number and content of nodes and edges and graph topologies. This continuous measure lends itself to grouping graphs using known clustering techniques.

To evaluate the usefulness of graph clustering we compare it to known non-graphical, microarchitecture dependent and independent information. We measure the effectiveness by applying and comparing the effect of *existing* optimizations to functions in clusters. We hypothesize that if functions in these clusters react homogeneously to optimization then they share common behaviors or characteristics.

The results of our evaluation on the SPEC benchmark suite show that clustering programs based on dynamic dataflow graphs produces much better clusters than clustering with instruction mixes (non-graphical) or other static/dynamic run time characteristics (non-graphical, microarchitecture dependent). We also qualitatively examined our best clusters to investigate if new distinct behaviors could be identified in the clusters. We were able to locate distinct behaviors in some clusters, but not all. This is likely because the SPEC benchmark is intended to be diverse so little obvious redundancy exists. Our results suggest that automated behavior identification may be possible with graphical clustering.

Interesting characteristics found with our techniques could assist in the discovery of a variety of optimizations in software, hardware or some hybrid of the two. These optimizations could be purely serial in nature or take advantage of parallelism. For instance, if one was to build clusters using data dependence information, resulting clusters might highlight data parallelism patterns ripe for optimization. As another example, we find an interesting pattern in our results (Table 5.4) which we call "guarded accessors". As we discuss later, this very common pattern could be further optimized.

The contributions of this chapter are: (1) A new method of approximately comparing the similarity of program graphs. (2) The application of this comparison method to approximately cluster program graphs. (3) A case study of the SPEC CPU 2006 benchmark suite using our new technique.

5.2 Methodology

Our clustering framework, CENTRIFUGE supports three primary steps: building program representations, clustering and evaluation. The first stage involves collecting static and dynamic characteristics of functions into approximate representations of the functions. The second stage involves two processes: comparison and clustering. Comparison involves examining the approximate representation of different functions to judge their similarity. This

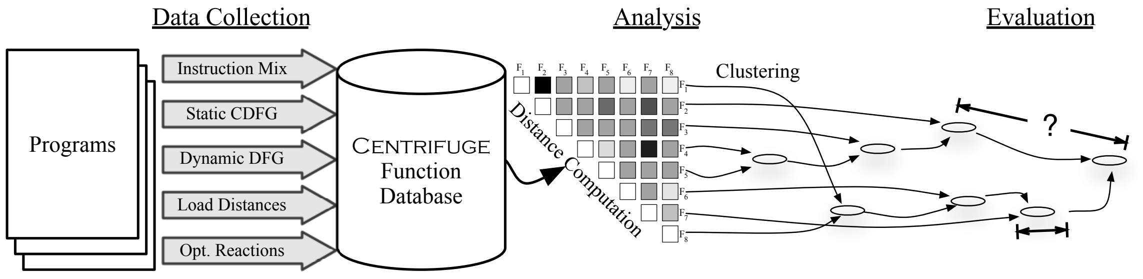


Figure 5.2: CENTRIFUGE involves three stages: data collection, analysis and evaluation. In data collection, information on function behavior is assembled. Analysis uses this information to compute distances between each function, in turn using this distance data to cluster the functions. Finally, in the evaluation stage we examine the clusters to determine their quality.

similarity information is then used to cluster the functions into similar groups. At the end of the second stage a user could study functions from each cluster. In the evaluation stage – the third and final stage – the contents of each cluster can be used to analyze the quality of the clusterings. This section provides details about each of these stages and is intended to give an overview of the processes, desiderata and tradeoffs relevant to each stage.

5.2.1 Representations

The first task in identifying code behaviors is to find static and/or dynamic features that capture function behavior. Good features will capture all characteristics that influence performance significantly. In this chapter, we include traditional static and dynamic characteristics that are believed to influence performance such as static instruction mix, static control/data flow graphs and dynamic data flow graphs. In addition we included two new features: dynamic data flow graphs augmented with locality information and a representation based on how well functions perform on existing hardware using existing optimizations. Each of these representations is described below.

1. Static Instruction Mix Static instruction mix can be useful as an indicator of how functions react to optimizations. For instance, clustering based on instruction mix may group math heavy functions or data movement functions in different clusters, and optimizations (such as vectorization or optimizations that use string movements) may operate on functions in different clusters differently. We compute the static instruction mix by compiling a program without any optimization and then categorizing the instructions into one of following categories: control transfer, arithmetic, logical, and memory. The fraction of instructions in each category are represented in a vector for each function in the code base.

2. Static CDFG Static control/data flow graphs are commonly used in compiler analyses. These graphs can capture the parallelism that is available in a function (or program) and thus may be good candidates to represent function behavior. As is common in compiler analyses, we use basic blocks as vertices in the flow graphs. We further annotate our graph vertices with basic block instruction mixes and edges with the dependence type (control, must pointer, may pointer, register). These dependencies were generated using the LLVM [100] compiler.

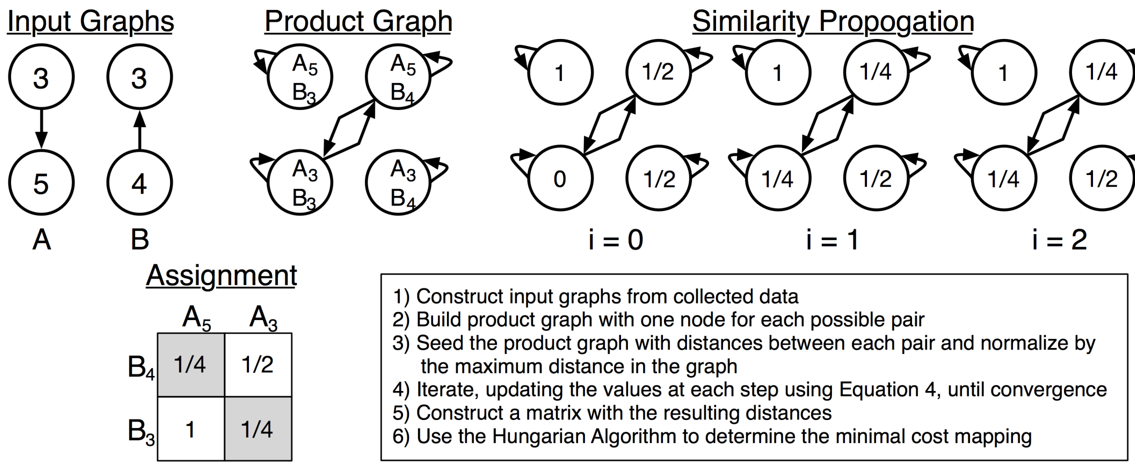
3. Dynamic DFG Static CDFGs include static memory dependence information. This dependence information is computed using alias analysis algorithms, which have been shown to be imprecise [114]. In the worst case, overly conservative analysis yields completely connected dependence graphs. Since there is only one complete graph given a number of vertices, this poor analysis effectively reduces the amount of information about the function. Dynamic data flow graphs, on the other hand, can be more detailed than Static CDFGs because they represent only *observed* dependence edges instead of potential dependence edges. Further, we can annotate these graphs with other dynamic information. In this chapter, we use dependence observation frequencies (the number of times a data dependence is observed between two basic blocks divided by the number of times the consuming basic block is executed) to add dependence edge weights to our Dynamic DFGs. To annotate these graphs' vertices, we compute the basic block's dependence chain length (the length of the longest producer-consumer chain of machine instructions within a basic block), number of integer instructions, number of floating point instructions, and number of memory loads, all of which are expressed as a fraction of total instructions. Dynamic features like execution count can also be used, though we have found that this can increase sensitivity to input bias.

4. Dynamic DFG with Load Distance It has been known for long time that Data locality is an important determinant of performance along with parallelism [9]. While the CDFG and Dynamic DFG are likely to be good at capturing parallelism constraints, they do not explicitly capture locality. To capture locality, we measure the number of dynamic instructions since each load's address was last accessed in the program (measured with PIN on executables with no optimizations). This is similar to reuse distance [42], except we count instructions instead of memory accesses. For example, if a called function accesses an address X which was last written by the calling function, a very low load distance will likely result. If, however, address X was last written during program initialization and has not been read since, a very high load distance will result, reflecting the improbability of the location being in the processor cache. We then compute the average and standard deviation of the logarithms of all load distances in a function and use these two measures as indicators of reuse.

5. Function Optimization Reactions In addition to the above representations, we can use existing optimizations to cluster similar functions. The intuition is that functions that are similar will be affected to the same degree when a set of optimizations are applied to them. Observing the optimization reactions, therefore, may give us an indirect indication of function characteristics. We construct this model by applying various combinations of known optimizations to a function, measuring the change in execution time for that function, and computing the function's optimization reaction, as defined in Eqn. 5.7. The optimization reactions are then used to cluster functions.

5.2.2 Comparison Methods

Now that we have the features to create the clusters, the next step is to determine how these features should be compared. We formalize the comparison by defining distance functions: for features that are represented in vector formats (*e.g.,* static instruction mix), we use the Euclidean distance between the two vectors to measure their similarity. In our initial experiments Euclidean distance yielded better results than alternatives (like Manhattan



	A_5	A_3
B_4	1/4	1/2
B_3	1	1/4

1) Construct input graphs from collected data
2) Build product graph with one node for each possible pair
3) Seed the product graph with distances between each pair and normalize by the maximum distance in the graph
4) Iterate, updating the values at each step using Equation 4, until convergence
5) Construct a matrix with the resulting distances
6) Use the Hungarian Algorithm to determine the minimal cost mapping

Figure 5.3: Simple example of the mapping stage, a variant of Similarity Flooding. In this example, we use unity edge weights and scalar (instead of vector) values in each vertex.

distance). For two vectors A and B both with length n, Euclidean distance is defined as:

$$\sqrt{\sum_{i=1}^{n}(A_i - B_i)^2} \tag{5.1}$$

Comparison of the graphical representations (*e.g.,* static CDFG, dynamic DFG, locality-annotated Dynamic DFGs) is more complex, deserving further explanation.

Traditional graph comparison algorithms such as isomorphism and subgraph isomorphism produce boolean "match" or "no match" results when two graphs are compared and are not useful for computing the *approximate* similarity between two functions represented as graphs. Instead, we use an approximate graph comparison technique [112] to compute the distance between two graphs. Determining the distance between two graphs involves two steps: mapping and scoring. In the mapping stage, we pair nodes from the two input graphs. That is, for each vertex in graph A, we find the vertex in graph B which is most similar, both in terms of the local information (basic block instruction mix) and neighborhood information (the similarity of connected vertices). The scoring phase then uses this mapping to compute vertex and graph structural similarity. With our graphical features, this process corresponds to finding similar basic blocks and then using the edge information (presence or absence of edge, and edge annotations) to judge the similarity of two functions.

Mapping The mapping phase has been adapted from the approach described by Melnik *et al.* as “Similarity Flooding” [112]. Although we use a similar set of steps, we have modified the ways in which values are normalized in various phases. The basic idea is to construct a new graph which contains all possible pairs of vertices and edges derived from edges in the two original graphs.[1] We assign a similarity value to each vertex (described shortly) in this graph and then iteratively propagate these values along edges. Upon convergence, each value represents the similarity of a pair of vertices which is sensitive to both the basic blocks' functional similarity (as determined by instruction mix distance) and the two graphs' structural similarity. We can then use these values to determine the optimal mapping. The precise details follow in addition to an example in Figure 5.3.

Formally, we begin by forming a tensor graph product (which we will use synonmously with “product graph”) of two function graphs to be matched. For weighted graphs (V_A, E_A) and (V_B, E_B) – wherein for (V, E), V is a set of vertices and E is a set of edge weights – we form the tensor graph product (V_P, E_P):

$$\begin{aligned} V_P &\equiv \{(A,B)|A{\in}V_A, B{\in}V_B\} \\ E_P &\equiv \{((A,B),(A',B'),W_A{\cdot}W_B)| \\ &\qquad (A,A',W_A){\in}E_A,(B,B',W_B){\in}E_B\} \end{aligned} \tag{5.2}$$

This product graph contains all the possible mappings, so the final mapping is intuitively a restricted subset of this graph.

Next, we define a primitive product graph vertex similarity function, $D(v)$. In our case, each product graph vertex represents a pair of basic blocks – one from each input graph – so we use the Manhattan distance between the basic blocks' vectors (the contents of which were described in Section 5.2.1. In our initial experiments, Manhattan distance yielded better results than the more popular Euclidean distance. For two vectors A and B each of length n, Manhattan distance is defined as:

$$\sum\nolimits_{i=1}^{n} |A_i - B_i| \tag{5.3}$$

Step three involves propagating the similarity scores along the graph edges, in a manner similar to that described by Melnik *et al.* [112]. This is an iterative calculation wherein

[1]This is known as a product graph. Specifically, we use the Tensor product graph.

at each step, i, we compute each vertex value M_j^i for each vertex j in V using the set of adjacent vertices, E_j, and weight of outgoing edges from j to each adjacent vertex a, W_j^a:

$$\begin{aligned} M_j^0 &= D(V_j) \\ M_j^i &= \frac{D(V_j)}{max(D(V_{i-1}))\cdot\sum W_j}\sum_{a\in E_j} W_j^a \cdot M_a^{i-1} \end{aligned} \tag{5.4}$$

The intuition behind this step is that if a node represents a good match and its neighbors are also good matches, then that node is probably a very good match and hence its score improves.[2]

The output from this similarity propagation step is a matrix of vertex-vertex pair scores which reflect both the primitive basic block and structural similarity. Determining the optimal mapping is thus a matter of selecting the set of pairs which are overall most similar. This is an instance of the assignment problem and we use the well known Hungarian Algorithm [95] to solve it.

Scoring Once the optimal mapping is established, we merge the two input graphs and score the merged graph. There are a number of scoring methods one could use. One might use the scores assigned by the previous mapping step. However, the scores from the mapping stage have been heavily affected by scores from potential pairs which turned out to be bad matches. (Recall that the product graph through which scores are propagated contains all possible pairs from which we select a small subset.) As a result, the scores from the mapping phase reflect the fitness of the selected vertex pairs relative to all the possible vertex pairs. The score we want, however, is the similarity of the two input graphs relative to other graphs.

We have designed an alternative scoring metric which essentially takes the basic block Manhattan distance and combines it with the structural mismatch which we define as the normalized quantity of unmatched edges. For each merged vertex, i in V, we produce the basic block instruction mix Manhattan distance, D_i, the number of common shared edges, S_i, and the number of non-shared edges, E_i. For vertices with no counterpart, we use $\sum_{d\in D_i} d$ for D_i and set $S_i = E_i$. We compute the score as:

[2]Detailed explanation of the Similarity Flooding algorithm is outside the scope of this chapter. Please refer to Melnik *et al.* [112] for more information.

$$\frac{1}{|V|}\sum_{i\in V}(D_i + \frac{E_i}{S_i + E_i}) \tag{5.5}$$

5.2.3 Clustering Methods

Equipped with quantitative comparisons for each representation, Centrifuge can use well known clustering methods such as agglomerative clustering, k-means, or SOM. In general, a clustering algorithm will attempt to group most similar functions together based on item distances or similarity of their features.

Some of these clustering techniques such as agglomerative clustering are hierarchical and produce dendrograms as the output instead of discrete clusters. The dendrograms can be converted to flat clusters by *thresholding* – cutting off growth of a cluster at a particular maximum diameter or a predetermined average internal distance. Thresholding creates a tradeoff between cluster size and the similarity of the contained functions. Higher thresholds typically create larger clusters with more dissimilarity than lower thresholds.

The user can also use an alternative property called savings instead of thresholding. *Savings* indicates the reduction in number of functions to be examined after clustering. Formally, the *savings* yielded from clustering f functions into c clusters is $(1 - \frac{c}{f})\cdot 100$. For example, if 50 functions are clustered 20 groups, then only 20 representative functions must be studied. This results in a 60% savings in the number of functions to be understood. The savings and threshold metrics are related because a larger threshold results in larger savings. Importantly, the savings metric can be compared across different representations and distance functions whereas threshold cannot since it is specific to the distance function. Additionally, parameterizing function organization via savings allows the user to gain better insight into functions' similarities by adjusting the savings level. Different clusters are shown at each savings level, indicating the distribution of similarity for various thresholds.

For this chapter, we use average linkage agglomerative clustering with the following four representations and distance functions:

1. Instruction Mix: Euclidean distance

2. Static CDFG: Similarity flooding-based graph distance defined in section 5.2.2 using

unity edge weights.

3. Dynamic DFG: Similarity flooding-based graph distance defined in section 5.2.2.

4. Locality annotated Dynamic DFG: We must combine the score from Similarity Flooding-based graph distance (defined in section 5.2.2) for the Dynamic DFG with the load distance information detailed in section 5.2.1. This data is represented by five numbers: the graph distance, g, for each function, the logarithms of the mean of the load distances, M_a and M_b along with logarithms of the standard deviations of the load distances, S_a and S_b. We combine them as such:

$$g + |M_a - M_b| + |S_a - S_b| \tag{5.6}$$

Our fifth representation, optimization reaction, also uses agglomerative clustering, but instead uses Ward's linkage [158]. Ward's linkage clustering attempts to minimize the variance of vector data within clusters. In other words, each time a cluster is selected for a particular data point, the cluster with the least increase in internal average variance is selected. Since our evaluation will eventually compute internal standard deviations, we can expect this clustering technique to create better clusters.

Optimization Reaction Metric Since profiling yields absolute times, we cannot directly compare optimization affects without some normalization of the times. Throughout this chapter, we use *optimization reaction*, defined below, to normalize all execution times to the range $[0, 1]$. If $T(f, o, i)$ is the runtime for function f when compiled with optimization o, executed with input set i and O is set of all profiled optimizations, we define the optimization reaction:

$$R(f, o, i) \equiv \frac{T(f, o, i) - min_{j \in O} T(f, j, i)}{max_{j \in O} T(f, j, i) - min_{j \in O} T(f, j, i)} \tag{5.7}$$

Normalizing the optimizations allows functions to be compared based on their relative effectiveness rather than raw speedup. The optimization reaction metric allows us to judge a reaction to an optimization by rank, so functions which are both sped up *best* by a particular optimization will have similar reactions for that optimization.

Parameter Tuning In all of these comparison methods, some information may be more important than others for similarity comparison *i.e.,* the number of memory instructions may influence performance more than the number of math instruction. To account for this, we can add additional parameters to each distance function allowing them to weight information differently. For instance, when we compute vector distances, we can instead use a weighted vector distance, changing the importance of each vector field. We add parameters for two representationss — Dynamic DFGs and Dynamic DFGs with Load Distances — in the following places:

1. **Vector Distances** All vector distance calculations (in both similarity flooding and scoring) get weights for each vector field.

2. **Similarity Flooding** In the mapping phase, similarity scores propagate through the product graph (Eqn. 5.2). At each iteration, these propogated scores are combined with local scores, creating a trade off between local and neighborhood similarity (Eqn. 5.4). We add weights to each input before adding them.

3. **Graph Scoring** After mapping, a merged graph is created and scored. During this scoring, several factors are combined to create a single similarity score. Here, we add importance weights when combining local and neighborhood similarity (similar to the last point), plus a weights to use for local and neighborhood similarity for unmatched vertices.

4. **Load Distances** When combining data flow graphs with locality information (Eqn. 5.6), we add three pieces of information: graph similarity, difference in log load distance means and difference in standard deviations. When tuned, each term is given a weight.

In total, we have 12 parameters which must be tuned. Given the extreme non-linearity of graph mapping, scoring and clustering, it is not possible to back-calculate optimal parameter values. Instead, we must use black-box optimization. In particular, we implemented simulated annealing, splitting our set of functions into $\frac{1}{3}$ for training and $\frac{2}{3}$ for testing. As an objective function, we measure the variance of reaction to optimization within the generated clusters. As such, simulated annealing with this objective function will attempt to minimize the variance of reaction to optimization within the generated clusters.

5.2.4 Evaluation Strategy for Centrifuge

We have now defined various representations, distance functions and clustering algorithms. Combined, these are used to create a clustering of functions. In this chapter, we compare functions across programs. This function granularity is neither the only possible granularity nor the optimal granularity, and virtually any block of code could be compared to any other. However, we chose function granularity for two reasons: First, functions are the granularity at which programmers generally think. Since we are attempting to find patterns in programmers' code, functions are a reasonable code unit. Second, identifying appropriate code sequences from an entire program is a difficult analysis problem. Other work [124, 151] explores this problem and should integrated into Centrifuge in the future.

To determine the utility of each representation, we must examine the resulting clustering and judge its quality. We can quantitatively measure quality by studying optimizations that have already been discovered; in particular, we wish to determine if our clustering could have been useful in discovering an optimization which already exists. If the functions in each cluster tend to react similarly to an existing optimization, then those clusters are significant with respect to that optimization. As a result, having those clusters may have been useful in developing the optimization. This metric can easily be quantified by measuring the functions' reactions to various optimizations. With this insight it is also fairly straightforward to come up with lower bound and upper bounds for the quality of the clusters. We can then compare various quality metrics about each representation's clustering to these bounds.

Random Clustering It is reasonable to expect a clustering to do no worse than randomly grouping functions. To estimate this worst case, we randomly generate similarity distances between all pairs of functions in the code base and use these distances along with average linkage agglomerative clustering to generate clusters. We would expect all properties of the functions in each of these clusters to be random selections of properties from the global set of functions.

Ideal Clustering Clusters built using the same information upon which they are evaluated should represent an upper bound. We will evaluate clusters based on the consistency with which their functions react to optimization, so we use the evaluation data and Ward's linkage agglomerative clustering to build this loose upper bound. (Loose due to the heuristic nature of agglomerative clustering in this context.) It is important to note that this is also an unfair upper bound because it assumes complete knowledge whereas other representations, by definition, are incomplete approximations of function behavior. With complete knowledge, this ideal clustering can account for random variations in runtime as well as measurement error.

5.3 Experimental Methodology

5.3.1 Data Collection

We collected static and dynamic execution characteristics data from functions in SPEC CPU 2006 suite. The static information (*viz.,* instruction mix and CDFG representations) were collected using a custom LLVM pass operating on code compiled to bitcode with "-O0 -g" options. Optimization is mostly turned off as we want to analyze code which is closely related to the original source code; optimizations distort this relationhip. We used the "mem2reg"[3] and "basicaa"[4] optimization and analysis passes before invoking our own. Dynamic data flow graphs and load distances were collected using custom PIN [107] tools.

Function Profiling Framework To measure the reactions of functions to optimization, we used performance counters via LiMiT (Chapter 3) to measure the execution times of functions precisely. To account for random variation, we execute programs three times and use the average.

[3]The mem2reg optimization is necessary to create sane LLVM bitcode from LLVM-GCC's output, which converts all stack variables to pointers instead of using LLVM's SSA form.

[4]Although LLVM has other alias analysis passes, we observed no difference in behavior.

Function Pruning Data is collected about SPEC functions from four different tools (*viz.,* LLVM, GCC, PIN and profiling tools), each of which have some limitations and caveats *e.g.,* measurement errors of very short functions *e.g.,* less than 10k cycles. To fairly evaluate our representations, we need to execute Centrifuge with a set of functions for which all of this data is available and accurate. As a result of pruning we are left with 628 functions, $\frac{1}{3}$ of which are selected for training and the remaining $\frac{2}{3}$ are used for evaluation in the following section.

5.3.2 Evaluation

To evaluate the merit of the Centrifuge methodology and the success of each representation we have proposed, we must judge the utility of its results. To do so, we take a retrospective approach: if the clusters which Centrifuge produces react homogeneously to *existing* optimizations, then there exists some relationship between the clusters and these existing optimizations. As a result, we speculate that these clusters *may* have been useful in discovering these optimizations.

Existing Optimizations We select four of GCC's optimizations (unswitch-loops, predictive-commoning, gcse-after-reload and tree-vectorize) which can be applied beyond the -O2 level. These are all of the optimizations which are turned on by '-O3' with the exception of inlining-based optimization, which we cannot use since we are measuring each function. We then create 15 different combinations of these four[5] and use them with -O2. We measured the effect of these fifteen different combinations of optimization flags on SPEC.

By evaluating Centrifuge against advanced optimizations (which often have minimal or negative effect on functions) and simultaneously combining them with basic optimizations we make our task both more difficult and more realistic. Newer optimizations are likely to be relatively complex and/or less-than-universally applicable as much of the "low-hanging fruit" has been realized in the simpler optimizations. (Indeed this is the case, as evidenced by the data in Table 5.1, which shows that our selected optimizations have significant

[5]GCC does not allow optimization re-ordering; each optimization can simply be on or off, so $2^4 - 1 = 15$ since we don't use only '-O2'.

	(Percent of Functions)	
	Speedup	Slowdown
Optimizations	≥ 15%	≥ 15%
-O2 -fpredictive-commoning -fgcse-after-reload	3.1%	6.9%
-O2 -fpredictive-commoning -ftree-vectorize	3.9%	9.1%
-O2 -fpredictive-commoning	2.7%	6.8%
-O2 -fgcse-after-reload	1.9%	9.6%
-O2 -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-vectorize	25.5%	4.4%
-O2 -ftree-vectorize	3.0%	7.5%
-O2 -fpredictive-commoning -fgcse-after-reload -ftree-vectorize	4.1%	9.0%
-O2 -fgcse-after-reload -ftree-vectorize	3.3%	10.2%
-O2 -funswitch-loops -fgcse-after-reload -ftree-vectorize	7.4%	7.5%
-O2 -funswitch-loops	5.0%	7.7%
-O2 -funswitch-loops -ftree-vectorize	33.2%	5.3%
-O2 -funswitch-loops -fpredictive-commoning -fgcse-after-reload	2.5%	8.0%
-O2 -funswitch-loops -fgcse-after-reload	2.5%	7.9%
-O2 -funswitch-loops -fpredictive-commoning	24.4%	3.3%
-O2 -funswitch-loops -fpredictive-commoning -ftree-vectorize	16.2%	9.1%

Table 5.1: Of our 628 selected functions, this table shows the percentage of programs that were sped up or slowed down a significant percent (15%) over optimization with just '-O2'.

effect on a small percentage of our experimental functions.) Should our representations work well only with simpler optimizations, they are less likely to be useful in the future. However, if our clustering is effective with advanced optimizations which are often not effective themselves, this implies that our method has very good resolution.

Additional Clusterings To provide context for the results, we artificially generate two additional clusterings – random and ideal – which represent loose lower and upper bounds. Random clusters are produced using a Gaussian random distance function to judge the distance between functions. The ideal clusters were built using the profiling data collected for evaluation. Since they are built and evaluated on the same data, they are close to optimal. This optimality is not guaranteed, however, due to the heuristic nature of agglomerative clustering. Additionally, we construct other clusters using randomly selected subsets of the evaluation data. These "Existing (1/2)" and "Existing (1/3)" clusterings use one-half and one-third of the optimizations, respectively, for construction but are evaluated on all of the optimizations. They are intended to judge how well optimization reaction data generalizes to other optimizations.

Savings In our cluster analyses, we parameterize groups of clusters by savings. Recall that "savings" indicates the reduction in number of functions to be examined after clustering and is defined as $(1 - \frac{c}{f}) \cdot 100$ for f functions grouped into c clusters. The savings level also represents a trade off: the higher the savings level, there are fewer clusters, but the functions in each cluster are less likely to be similar. At the extreme high end, all functions are in a single cluster, and thus there is no interesting information. At the extreme low end, each cluster contains one function, so there is also no interesting information. The Centrifuge user must determine an appropriate point.

5.4 Results

We design statistical tests to answer the following two questions and qualitatively evaluate two more:

1. Are similarity distances (as determined by each representation) within random and

ideal clusters different from global similarity distances? If a representation captures characteristics pertinent to the optimizations, we expect that representation to produce small distances between functions in ideal clusters. As a sanity check, we expect distances within random clusters to be little different from the global set of distances. (Section 5.4.1)

2. How consistently does optimization affect functions in each cluster? We would expect functions in clusters which are relevant with respect to existing optimizations to react similarly to optimization. (Section 5.4.2)

3. After tuning, what features proved most effective? (Section 5.4.3)

4. Qualitatively, what type of clusters are produced? (Section 5.4.4)

5.4.1 Evaluating Distances in Ideal and Random Clusters

We first wish to determine if our representations' judgment of similarity is interesting with respect to existing optimizations. This test uses each representation's distance function but not the clusters produced using these distances, only the random and ideal clusters. As such, it allows us to test for significance using a minimal amount of clustering and thus eliminate a potential source of error.

For each representation, we compute the average distance between all function pairs. This average global distance indicates how far apart – on average – functions usually are for each representation. We then examine the ideal and random clusterings at each savings level (0%-50%). For each cluster, we look at the distances between each pair of functions, as determined by each representation. We compute an average of these internal distances and subtract that number from the representation's global average. If the cluster is significant with respect to the representation, the cluster's internal distance average will be small and thus this difference will be high. We also run Student's T-Test [144] to determine the significance of this difference. We further calculate weighted (by cluster size) averages of the mean differences and average T-Test probabilities for each savings range.

The summary results of this analysis are shown in Figure 5.4. As expected, differences in

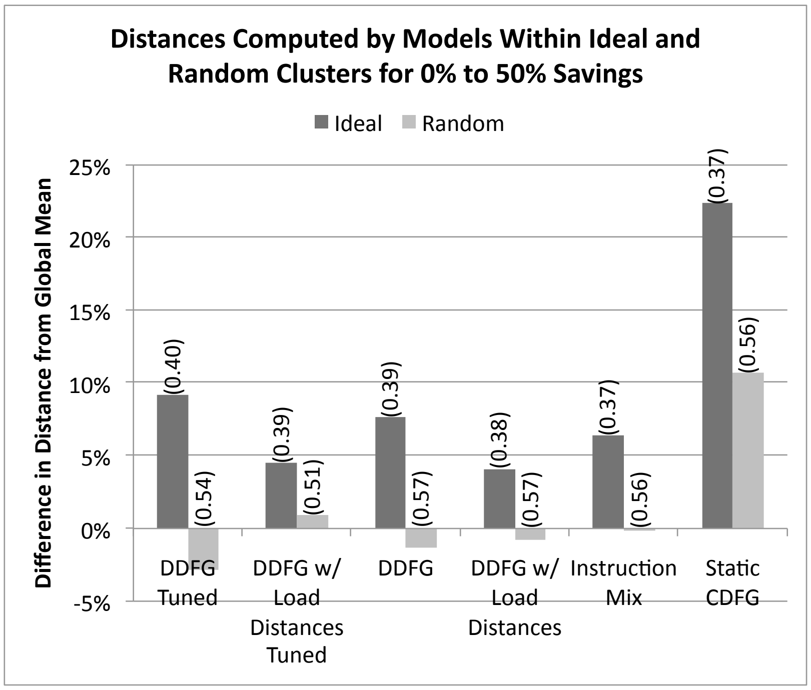


Figure 5.4: To measure the acuity of our representations we compare the average distance between clustered functions (in the 0% to 50% range) to the global average distance. We expect the random clusters' averages to be approximately zero and the ideal clusters' averages to be greater than that. Measurements show that random clusters tend to be zero or negative whereas ideal clusters have high differences (T-Test significance values are in parenthesis). Results indicate that our representations' distances are not correlated with random clusters, but are correlated with ideal clusters.

the random clusters tend to be closer to zero than in the ideal clusters.[6] This result indicates that there is no correlation between random clusters and our representation distances. The fact that differences in ideal clusters are higher indicates that representation distances tend to be smaller in these clusters than the average global distance. Further, the T-Test probability values are smaller for the ideal clusters, showing a greater chance that these differences are significant. On the whole, this test shows that all of the proposed representations have a closer relationship to ideal clusters than random clusters. Although tempting, we cannot determine from this difference data which of the representations is superior. Although normalized using distance averages, these distances are not guaranteed to have similar distributions, so we cannot determine the significance of these values relative to each other.

5.4.2 Clustering Quality and Implications

Next, we evaluate the consistency of optimization reactions within each cluster. In contrast to the previous test of distances in random and ideal clusters, this test evaluates the clusters generated by each of our representations in addition to the two artificial ones – random and ideal – allowing us to directly compare all of them. To test this consistency, we compute the standard deviation of reaction to optimization (as defined in Eqn. 5.7) for all functions within each cluster. The intuition behind this metric is simple: good clusters should contain functions which react similarly to optimization. To present these data, we compute the reaction standard deviation for each cluster and average across all the clusters at each savings level.

Figure 5.5 shows the consistency of optimization reaction for each representation described in this chapter, plus the ideal and random clusters. As expected, the consistency of the clusters tends to decrease with savings because the clusters must grow in size, forcing functions with decreasing similarity into the same clusters. This is a direct result of the tradeoff discussed in the above "savings" paragraph.

[6] In some cases, the bars for random clusters are visibly above or below zero. This is because the distributions are skewed, so random selection is more likely be closer to the global median rather than the global average. The more important comparison is the difference between the two bars.

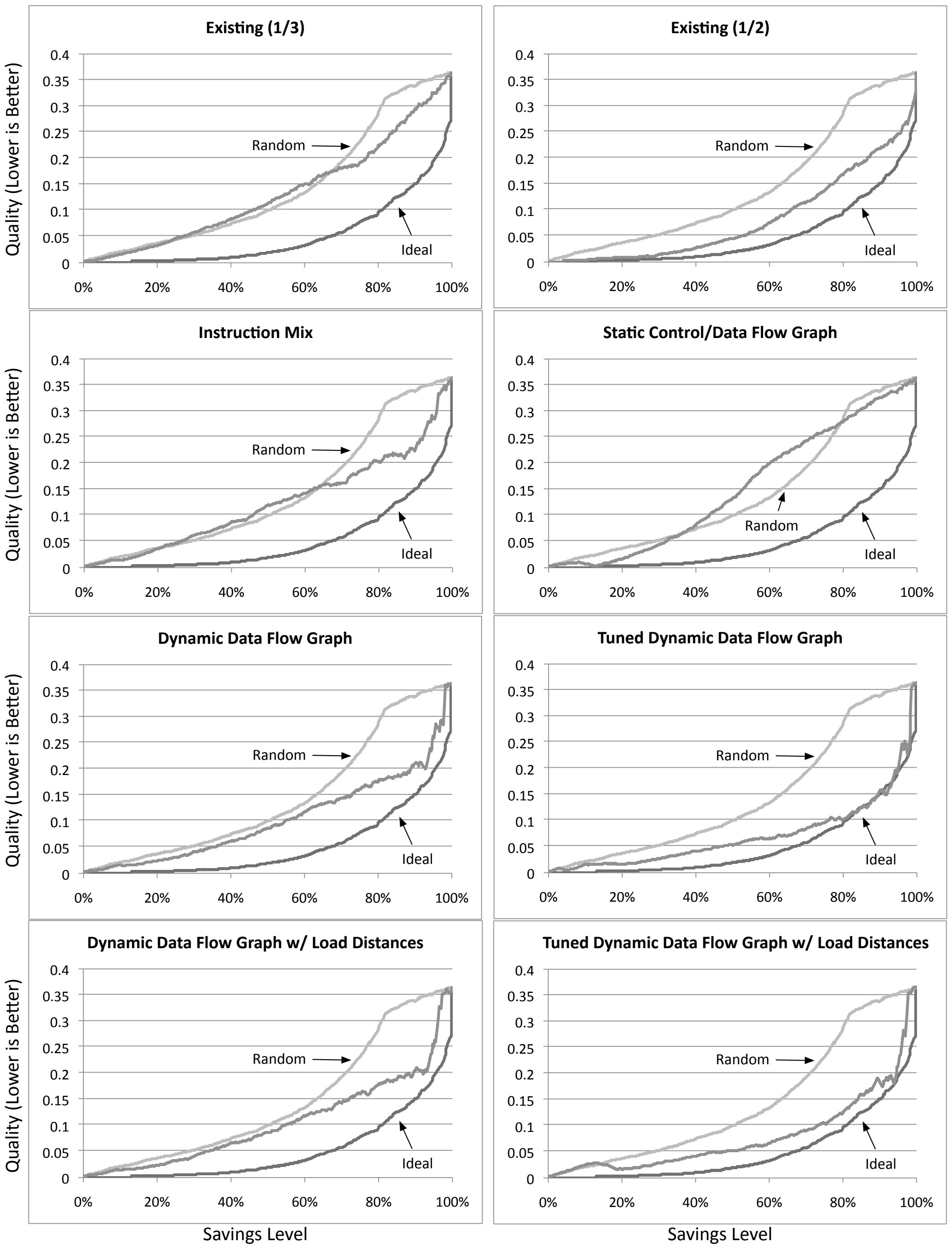


Figure 5.5: Consistency of optimization reactions for clusters from various representations. Consistency is calculated as standard deviation, so lower numbers are better. Each clustering is shown relative to loose upper and lower bounds, "ideal" and "random". Although none of the representations perform perfectly, we see that "Tuned Dynamic Data Flow Graph" performs very well.

There are several other interesting things to note in these results:

1. These results clearly demonstrate that instruction mixes and "Existing (1/3)" are largely worthless for clustering because they are barely better than random.

2. As expected, using a substantial amount of the evaluation data (Existing 1/2) produces good results. When this quantity is decreased to one-third, however, the results are little better than random. This result implies that optimization reactions themselves are poor predictors of similarity because they do not generalize to other optimizations.

3. Few of the representations perform well in the very low savings range (0% to 25%) compared to ideal. Although subtle, this affirms a widely-held belief that performance is extremely difficult to predict accurately. Although several of the representations are able to predict large performance changes, none can do so at the accuracy required to perform well in this regime.

4. Static CDFG fairs very poorly overall but perform well in the small savings range (0% to 25%). This result implies that Static CDFGs are very useful for identifying identical functions, but poor for gauging approximate function similarity. We speculate that this is due to weaknesses in alias analysis and as a result, Static CDFGs contain too many edges to be useful.

Overall, our proposed representations are a mixed bag: some perform well and some do not. In general, representations utilizing dynamic data flow graphs perform well; the area under the tuned dynamic data flow graph representation curve is 80% closer to the area under the ideal curve than that of the random curve! This result is encouraging and confirms the utility of CENTRIFUGE: clusters of functions which react similarly to optimization can be built using generic representations.

Discussion of Optimizations We can also examine our results' relationship to the optimizations we are using for evaluation – predictive-commoning, tree-vectorize, unswitch-loops, and gcse-after-reload. Predictive commoning examines loops and attempts to pull out redundant computations. Tree vectorization is GCC's auto vectorizer, intended to create parallel SSE code from normal loops. Unswitch loops pulls conditional statements out of loops allowing the loop to be optimized with less hindrance. Finally, GCSE after reload invokes another subexpression elimination with the goal of eliminating redundant loads. In each case, there are characteristics in the code which determine whether or not the optimization can be applied and how performance will be affected. The results of some of our representations imply that they are capturing some of these characteristics whereas others are not.

With regard to our DDFG (dynamic data flow graph) representation, what features relevant to these optimizations could it be capturing? All of these optimizations deal with movement (or elimination) of operations and thus the optimizations sensitive to both control and data dependences – if certain dependencies exist, the optimization cannot be applied. Alternatively, if certain dependence patterns do exist, the first three optimizations may be applied and strongly affect performance.

We also add load distances to indicate data locality for one representation. Although it does not seem to have a strong effect on our results, it is possible that it could related strongly to unswitch loops and GCSE optimizations. Both eliminate redundant operations (which are likely to have memory operations); since these operations are likely to occur often and are guaranteed to access the same memory, they will likely have very good locality. As such, low load distances may weakly indicate that these optimizations would apply.

Lastly, it is not surprising that instruction mix does poorly; merely knowing that a function has memory loads or integer calculations tells us nothing about whether or not the operations can be moved.

Although our results are specific to these four optimizations, we suspect that dependencies are key to most complex optimization, thus our DDFG representations are likely to scale well.

5.4.3 Evaluation of Parameter Tuning

As discussed in Sec. 5.2.3, "Parameter Tuning", we use simulated annealing to optimize parameters used for our graph comparisons – dynamic data flow graphs and dynamic data

Parameter	DDFG	DDFG w/ LD
Basic Block Annotations		
Dependence Chain Length	1.0	0
Number of Int Instructions	1.0	0.68
Number of FP Instructions	0.64	0.95
Number of Memory Loads	0	0.26
Graph Mapping		
Neighborhood Score (vs. local similarity score)	0.46	1.0
Graph Scoring		
Difference in Local Similarity	0.05	0
Difference in Edges	1.0	0.54
Unmatched Annotation Penalty	0	0.18
Unmatched Edge Penalty	0.64	0.33
Load Distances		
Graph Similarity Score	-	0.45
Difference in LD Means	-	0.84
Difference in LD Std. Dev.	-	0

Table 5.2: Parameter values for dynamic data flow graphs and dynamic data flow graphs with load distances after tuning them via simulated annealing. Area under the curves of Fig. 5.5 was used as the objective function to minimize.

flow graphs with load distances (locality information). The resulting weights are shown in Table 5.2. Though potentially interesting, these results have no guarantee of optimality as they are found in continuous in 12-dimensional space. Although not optimal these weights were used to generate the clusters evaluated and presented here and evaluate better than untuned representations. There are several interesting observations to make about the weights:

1. The dependence chain length within a basic block is hugely important to gauging similarity unless load distances are being considered. Further, the number of memory loads in each basic block are not a good indicator of similarity unless load distances are also considered.

2. Although the similarity between basic blocks is used in the graph mapping stage, they are not used in scoring. Instead, the graphical edge similarity (number of matched/unmatched edges) is used. As a result, graphs of different sizes are penalized far less with out tuned models than untuned.

3. When integrating load distances, the difference in standard deviations is not used, however the difference in log load distance averages is given nearly twice the weight as the graph similarity. This indicates that data locality is important in judging similarity.

5.4.4 Qualitative Evaluation of Cluster Results

The results presented in Figure 5.5 show that tuned dynamic data flow graphs tend to do a reasonable job clustering functions which will react similarly to optimization. So, what sort of functions get grouped together? To answer this question, we examine clusters produced around the 15% to 20% savings level. In some clusters, we see functions which are nearly duplicates. Others have obvious patterns – these *may* indicate potential widely-applicable opportunities for optimization. Other clusters contain functions which do not appear similar, yet react similarly to optimization and have similar data flow graphs. Here are some examples.

Cluster #1

```
rtx gen_rtx_fmt_ee (RTX_CODE code, enum machine_mode mode,
                    rtx arg0, rtx arg1) {
    rtx rt = ggc_alloc_rtx (2);
    memset (rt, 0, sizeof (struct rtx_def) - sizeof (rtunion));
    PUT_CODE (rt, code); PUT_MODE (rt, mode);
    XEXP (rt, 0) = arg0; XEXP (rt, 1) = arg1;
    return rt;
}

rtx gen_rtx_fmt_e0 (RTX_CODE code, enum machine_mode mode, rtx arg0) {
    rtx rt = ggc_alloc_rtx (2);
    memset (rt, 0, sizeof (struct rtx_def) - sizeof (rtunion));
    PUT_CODE (rt, code); PUT_MODE (rt, mode);
    XEXP (rt, 0) = arg0; X0EXP (rt, 1) = NULL_RTX;
    return rt;
}

rtx gen_rtx_fmt_s (RTX_CODE code, enum machine_mode mode, rtx arg0) {
    rtx rt = ggc_alloc_rtx (2);
    memset (rt, 0, sizeof (struct rtx_def) - sizeof (rtunion));
    PUT_CODE (rt, code); PUT_MODE (rt, mode);
    XSTR (rt, 0) = arg0;
    return rt;
}
```

Table 5.3: An example cluster of near-duplicate functions. These functions are all from 403.gcc.

Near-Duplicates Programs like GCC tend to have many functions which are auto-generated and thus nearly identical, like those shown in Table 5.3. Many of these near-duplicates are very small functions (often constructors) with a single basic block. These clusters are largely uninteresting as they are best optimized via inlining and in-context optimization.

Guarded Accessors By far, the most common pattern we see clustered is a pattern we call "guarded accessors". Although class field accessors are thought to be a pattern used primarily in object-oriented languages, we also see in C that many functions are created to conditionally get or mutate a data structure. For an example, see Table 5.4. These clusters represent a common behavior that one might be able to optimize. First, all of them have a very uncommon branch case – the error conditions – where performance does not matter. Second, the conditions being checked have few side effects (with the exception of NULL checks), so they can be evaluated in any order. These uncommon cases may be detected via profiling. Alternatively, it may be reasonable to assume simple return values like -1 or not returning (like the exit call) are uncommon cases. Further, the functions shown here are not directly affected much by the four optimizations we are applying, and thus may represent a new optimization opportunity.

Non-Intuitive Clusters There is another set of clusters which contain functions which react similarly to optimization, but it is not intuitively (or obviously) clear why. Table 5.5 shows an example – one function which reverses a list and another that inserts a record into a hash table. While one has a loop (which is likely not unrolled as it is pointer chasing) the other has no loop in either its body nor function calls. One allocates memory and makes a function call, the other is a terminal in the call chain. What do these functions have in common? First, their dynamic dataflow graphs are identical (as shown in Figure 5.6), have very similar edge weights and their basic block annotations are similar – they have very few integer and floating point calculations but have memory loads in some basic blocks. The similar memory patterns mean that similar data placements, layouts or prefetching strategies may work similarly on both functions. This class of clusters is probably the most interesting; it shows similarity that likely would not have been recognized during manual

Cluster #2

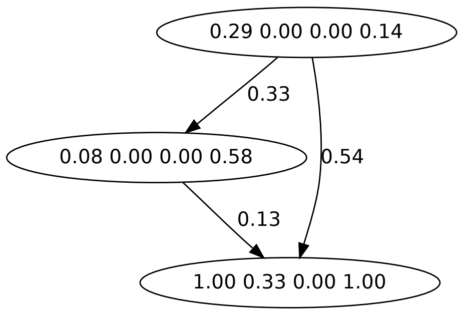
```
int CCTK_NumTimeLevelsFromVarI (int var) {
    return ((0 <= var && var < total_variables) ?
        groups[group_of_variable[var]].n_timelevels : -1);
}


int CCTK_GroupTypeFromVarI (int var) {
    return ((0 <= var && var < total_variables) ?
        groups[group_of_variable[var]].gtype : -1)


int ETree_frontSize (ETree *etree, int J) {
    if ( etree == NULL || J < 0 || J >= etree->nfront ) {
        fprintf(stderr, "\n fatal error in"
                        "ETree_frontSize(%p,%d)\n"
                        " bad input\n", etree, J);
        exit(-1);
    }
    return(etree->nodwghtsIV->vec[J]);
}
```

Table 5.4: An example cluster in which all functions are guarded accessors. The first two are from 436.cactusADM and the last is from 454.calculix. In these clusters, conditions are checked before returning data from a structure. The error case, however, is different – two return error codes, one aborts. Error cases are uncommon, so they have little effect on performance, and may provide an (de)optimization opportunity.

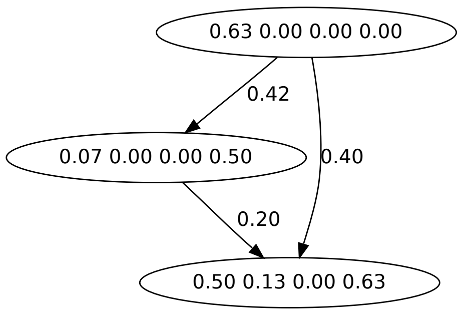
Cluster #3

```
tree nreverse (tree t) {
    tree prev = 0, decl, next;
    for (decl = t; decl; decl = next) {
       next = TREE_CHAIN (decl);
       TREE_CHAIN (decl) = prev;
       prev = decl;
    }
    return prev;
}

void type_hash_add (unsigned int hashcode, tree type) {
    struct type_hash *h;
    void **loc;

    h = (struct type_hash *) ggc_alloc (sizeof (struct type_hash));
    h->hash = hashcode;
    h->type = type;
    loc = htab_find_slot_with_hash(type_hash_table, h, hashcode, insert);
    *(struct type_hash **>) loc = h;
}
```

Table 5.5: An example cluster in which functions react similarly to optimization, but do not appear similar. Despite the dissimilarity, these two functions have identical dynamic data flow graphs and react similarly to optimization.



(a) `nreverse`



(b) `type_hash_add`

Figure 5.6: Dynamic data flow graphs for the cluster shown in Table 5.5. Althought the code snippets do not appear similar, the resulting DDFGs are very similar.

inspection of the code, yet our generic representation and profiling of optimized code shows similar behavior and reactions to optimization.

5.5 Related Work

Related research topics include performance prediction mechanisms, machine learning applications in optimization, program behavior analysis and code mining for topic analysis and microarchitectural enhancement, and computational kernel classification.

Several works [26, 44, 73] attempt to predict the reaction of code to various program transformations using code features, profiling information from subsets of possible program transformations and dynamic program characteristics (like instruction mix and strides), respectively. These works, however, are based entirely on overall program speedup and whole program analysis. While there are some similarities to our work (in spirit) none of these works discuss clustering based on the program features. These works are largely motivated by the problem of determining if/when an optimization should be applied during compilation and not for characterizing program behavior.

A large body of work [1, 4, 27, 60, 102, 139, 140] attempts to apply machine learning techniques to compiler optimizations. The bulk of this work attempts to improve existing optimizations and their associated heuristics via machine learning or find better combinations of program optimizations (phase ordering). While some of them implicitly use clustering, none of these cluster on graphical formats which this work shows to be advantageous.

The software engineering community has several works [94, 147, 152] in which functions are clustered to identify functions which have similar keywords or are semantically similar. These efforts use textual analysis, so there is little reason to believe these analysis techniques could be relevant to program characterization as minor changes in source code (*e.g.,* changing variable names) do not typically affect optimization or performance behavior.

The software engineering community has also long worked to identify "copy and paste" code. Many approaches [13, 14, 62, 84, 92, 93, 105, 128, 137] have been developed, nearly all of which rely solely on static code analysis. Typically, these tools are not designed to calculate similarity – they only detect when code has been copied, thus any graph matching algorithms they use do not require the same level of approximation our approach provides.

In the architecture community there is also some relevant work in both benchmark selection and program graph mining. Several papers [47, 83, 129] use analysis to find redundancy in sets of benchmark programs. These techniques can be used for benchmark selection; however they operate at the granularity of an entire program. As a result, this work is largely complementary and in fact was indirectly used to select benchmarks for this chapter as [129] was used to create SPEC06. Another set of papers [34, 35, 72] use program mining to assist in instruction set customization. In this work, Clark *et al.* examine and find common patterns in graphs; however their techniques work to find very small patterns – several instructions – only rather than function-granularity patterns and idioms.

Another effort to recognize patterns in code is XARK [7]. XARK's is able to classify loop structures into several categories of computational kernels types such as inductions, maps and scalar assignments. CENTRIFUGE is distinctly different as it computes approximate similarity and hierarchically clusters functions. Other work [64] on design pattern mining uses inexact graph matching, an approach similar to ours. It uses a different approximate graph matching algorithm, however, and operates on UML graphs rather than automatically collected data.

5.6 Conclusion

In this chapter, we proposed clustering on graphical intermediate program representations for program characterization. We introduced a novel approximate graph similarity metric to drive our graph clustering. Unlike existing approaches, we avoid some feature selection, operating directly on graphs. To evaluate the effectiveness of our approach, we designed a framework called CENTRIFUGE that clusters functions based on common static and dynamic characteristics. We have shown that functions grouped by graphical properties tend to react similarly to several existing optimizations. These results indicate that (1) it is possible to classify code snippets into behavioral groups which react similarly to optimization and (2) that clustering on graphical representations produces better results compared to static non- graphical formats. Further, based on manual analysis of some of clustered functions we determine that there is potential for discovering interesting patterns and thus our techniques may be useful for qualitative program characterization. Future work will do two things: First, it is likely that our approximate graph matching technique can be improved further. Second, there are a number of other interesting applications like finding and automatically synthesizing accelerators.

Chapter 6

Machine Learning Architectural Behaviors for Malware Detection

The proliferation of computers in any domain is followed by the proliferation of malware in that domain. Systems, including the latest mobile platforms, are laden with viruses, rootkits, spyware, adware and other classes of malware. Despite the existence of anti-virus software, malware threats persist and are growing as there exist a myriad of ways to subvert anti-virus (AV) software. In fact, attackers today exploit bugs in manually-written AV software to break into systems.

In this chapter, we examine the feasibility of automatically building a malware detector based on microarchitectural behavior data produced by malware. We find that data from performance counters can be used to identify malware and that our detection techniques are robust to minor variations in malware programs. As a result, after examining a small set of variations within a family of malware on Android ARM and Intel Linux platforms, we can detect many variations within that family. Further, we propose hardware modifications allow the malware detector to run securely beneath the system software, thus setting the stage for AV implementations that are simpler and less buggy than software AV. Combined, the robustness and security of hardware AV techniques have the potential to advance state-of-the-art online malware detection.

6.1 Introduction

Malware – short for malicious software – is everywhere. In various forms for a variety of incentives, malware exists on desktop PCs, server systems and even mobile devices like smart phones and tablets. Some malware pollutes devices with unwanted advertisements, creating ad revenue for the malware creator. Others can dial and text so-called "premium" services resulting in extra phone bill charges. Some other malware is even more insidious, hiding itself (via rootkits or background processes) and collecting private data like GPS location or confidential documents.

This scourge of malware persists despite the existence of many forms of protection software, antivirus (AV) software being the best example. Although AV software decreases the threat of malware, it has some failings. First, because the AV system is itself software, it is vulnerable to attack. Bugs or oversights in the AV software or underlying system software (*e.g.,* the operating system or hypervisor) can be exploited to disable AV protection. Second, production AV software typically use static characteristics of malware such as suspicious strings of instructions in the binary to detect threats. Unfortunately, it is quite easy for malware writers to produce many different code variants that are functionally equivalent, both manually and automatically, thus defeating static analysis easily. For instance, one malware family in our data set, AnserverBot, had 187 code variations. Alternatives to static AV scanning require extremely sophisticated dynamic analysis, often at the cost of significant overhead.

Given the shortcomings of static analysis via software implementations, we use dynamic analysis of programs to detect malware. We posit that dynamic analysis makes detection of new, undiscovered malware variants easier. The intuition is as follows: we assume that all malware within a certain family of malware, regardless of the code variant, attempts to do similar things. For instance, they may all pop up ads, or they may all take GPS readings. As a result, we would expect them to work through a similar set of program phases, which tend to exhibit similar detectable properties in the form of performance data (*e.g.,* IPC, cache behavior).

In this chapter, we pose and answer the following central feasibility question: *Can dynamic performance data be used to characterize and detect malware?* We collect longitudinal, fine-grained microarchitectural traces of recent mobile Android malware and Linux rootkits on ARM and Intel platforms respectively. We then apply standard machine learning classification algorithms such as KNN or Decision Trees to detect variants of known malware. Our results indicate that relatively simple classification algorithms can detect malware at nearly 90% accuracy with 3% false positives for some mobile malware.

6.2 Background on Malware

In this section, we provide an abbreviated and fairly informal introduction on malware.

6.2.1 What is Malware and Who Creates It?

Malware is software created by an attacker to compromise security of a system or privacy of a victim. A list of different types of malware is listed in Table 6.1. Initially created to attain notoriety or for fun, malware development today is mostly motivated by financial gains [24, 142]. There are reports of active underground markets for personal information, credit cards, logins into sensitive machines in the United States, etc. [150]. Also, government-funded agencies (allegedly) have created sophisticated malware that target specific computers for espionage or sabotage [31, 97, 98]. Malware can be delivered in a number of ways. To list a few, an unsuspecting user can be tricked into: clicking on links in "phishing" emails that download and install malware, opening email attachments with malicious pdfs or document files, browsing web pages with exploits, using infected USB sticks or downloading illegitimate applications repackaged to appear as normal applications through mobile stores.

6.2.2 Commercial Malware Protections

The most common protection against malware is anti-virus (AV) software. Despite what the name anti-virus suggests, anti-virus can also detect and possibly remove categories of malware besides viruses. A typical AV system works by scanning files during load time for known signatures, typically code strings, of malware. Figure 6.1 shows how anti-virus signatures are prepared: Honeypots collect malware and non-malware which are then analyzed by humans to create signatures. These signatures are then delivered to the host anti-virus

Table 6.1: Categories of Malware

Malware	**Brief Description**
Worm	Malware that propagates itself from one infected host to other hosts via exploits in the OS interfaces typically the system-call interface.
Virus	Malware that attaches itself to running programs and spreads itself through users' interactions with various systems.
Polymorphic Virus	A virus that, when replicating to attach to a new target, alters its payload to evade detection, *i.e.* takes on a different shape but performs the same function.
Metamorphic Virus	A virus that, when replicating to attach to a new target, alters both the payload and functionality, including the framework for generating future changes.
Trojan	Malware that masquerades as non-malware and acts maliciously once installed (opening backdoors, interfering with system behavior, etc).
AdWare	Malware that forces the user to deal with unwanted advertisements.
SpyWare	Malware that secretly observes and reports on users computer usage and personal information accessible therein.
Botnet	Malware that employs a user's computer as a member of a network of infected computers controlled by a central malicious agency.
Rootkit	Malware that hides its existence from other applications and users. Often used to mask the activity of other malicious software.

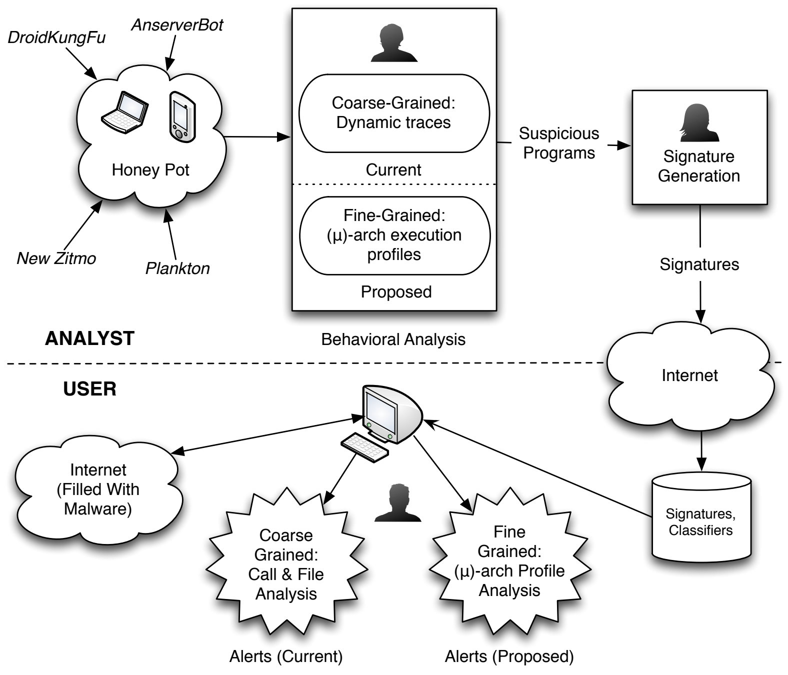


Figure 6.1: AV signature creation and deployment.

software periodically.

A complementary approach to signature-based detection is also used in practice [131]. In reputation based AV detection, users anonymously send cryptographic signatures of executables to the AV vendor. The AV vendor then determines how often an executable occurs in a large population of its users to predict if an executable is malware: often, uncommon executable signatures occurring in small numbers are tagged as malware. This system is reported to be effective against polymorphic and metamorphic viruses but does not work against non-executable threats such as malicious pdfs and doc files [19]. Further it requires users to reveal programs installed on their machine to the AV vendor and trust the AV vendor not to share this secret.

6.2.3 How Good is Anti-Virus Software?

Just like any other large piece of software, AV systems tend to have bugs that are easily exploited, and thus AV protections are easily bypassed. In a recent paper, Jana and Shmatikov [80] found that all of the 36 commercially available AV systems they examined could be bypassed. Specifically, they detected many bugs in the code that parse program binaries which either allowed bad code to pass undetected or gain higher privilege. They argued that the problem of building robust parsers (and hence software malware detectors) is not easy since the number of file formats is quite large, and many of their specifications are incomplete in several ways. Their paper demonstrates the futility in trying to secure complex, million-line softwares like AV. Unlike software detectors, the hardware malware detectors we propose do not have to deal with multiple executable formats. Instead they work on single input format – integer streams from performance counters. Further, they are not easily turned off. Thus hardware detectors are significant step towards more robust detectors.

6.2.4 Malware Arms Race

There is an arms race between malware creators and detectors. The earliest detectors simply scanned executables for strings of known bad instructions. To evade these detectors, attackers started encrypting their payloads. The detectors, in response, started scanning for the decryption code (which could not be encrypted) packed with the malware. The malware creators then started randomly mutating the body of the payload by using different compilation strategies (such as choosing different register assignments or padding NOPs) to create variants [146].

In response to these advances in malware creation, defenders were motivated to consider *behavioral* detection of malware instead of static signatures. Behavior-based detection characterizes how the malware interacts with the system: what files it uses, the IPC, system call patterns, function calls and memory footprint changes [32, 57, 99]. Using these characteristics, detectors build models of normal and abnormal program behaviors, and detect abnormal execution by comparing against pre-built behavioral models. Many of these schemes use machine learning techniques to learn and classify good and bad behaviors from labeled sets [12, 15, 103, 132].

6.2.5 Improving Malware Detection

While behavioral schemes permit richer specification of good and bad behaviors than static checkers, they tend to have high performance overheads since the more effective ones demand creation and processing of control- and data-flow graphs. Because of their overheads behavior-based detectors are not typically used on end hosts, but analysts in malware-detection companies may use them to understand malware-like behaviors. All of these techniques are envisioned to be implemented in software.

In this work, for the first time, we use hardware performance counters for behavior based detection of malware, and describe the architecture necessary to support malware detection in hardware. Our performance counter based technique is a low-overhead technique that will not only allow analysts to catch bad code more quickly, it may also be feasible to deploy our system on end hosts. Unlike static signature based detection AV, we aim to detect variants of malware from known malware signatures. Unlike reputation based system our scheme does not require users to reveal programs installed on their computer.

Recent research has also examined using hardware performance counters for detecting anomalous program behaviors [108, 160]. This is a different and (intuitively) harder problem than attempted here. The anomaly detection works aim to detect small deviations in program behavior during an attack such as a buffer overflow or control flow deviation from otherwise mostly benign execution. In contrast, we attempt to identify execution of whole programs such as key logger when it is run, typically as the end result of exploitation such as buffer overflow vulnerability.

6.3 Key Intuition

A major thesis of this chapter is that runtime behavior captured using performance counters can be used to identify malware and that the minor variations in malware that are typically used to foil signature AV software do not significantly interfere with our detection method.

The intuition for this hypothesis comes from research in program phases [78, 134]. We know that programs exhibit phase behavior. They will do one activity A for a while, then switch to activity B, then to activity C. We also know that programs tend to repeat these

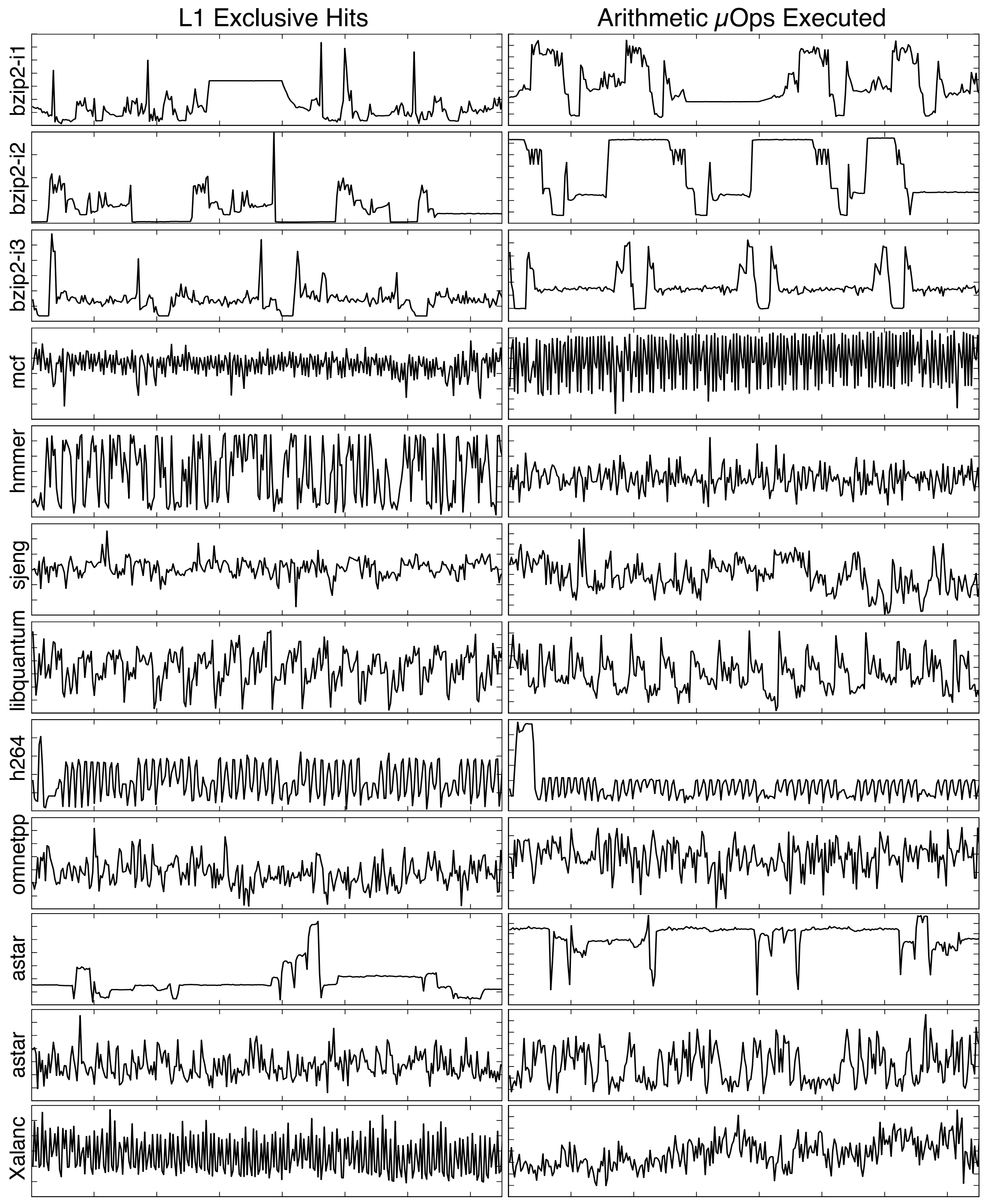


Figure 6.2: Performance counter measurements over time in the SPEC benchmark suite. We also observe readily apparent visual differences between the applications. Intuitively, we expect it to be possible to identify programs based on these data.

phases – perhaps the program alternates between activities B and C. Finally, and most importantly, it has been shown that these phases correspond to patterns in architectural and microarchitectural events.

Another important property of program phases and their behaviors is that they differ radically between programs. Figure 6.2 plots event counts over time for several SPEC applications. In it, we see the differences between the benchmarks as well as interesting phase behavior. Given these data, it seems intuitive that these programs could be differentiated based on these time-varying signatures.

Our hypothesis that minor variations in malware do not significantly affect these data cannot be inferred from previous work. Rather, it is based on two observations:

• First, regardless of *how* malware writers change their software, its semantics do not change significantly. For instance, if a piece of malware is designed to collect and log GPS data, then no matter how its writer re-arranges the code, it still collects and logs GPS data.

• Second, we assume that in accomplishing a particular task there exist subtasks that cannot be radically modified. For instance, a GPS logger will always have to warm up the GPS, wait for signals, decode the data, log it and at some future point exfiltrate the data out of the system. As a result of these invariant tasks, we would expect particular phases of the malware's execution to remain relatively invariant amongst variations.

If indeed microarchitectural behaviors remain relatively invariant amongst variations *and* they are unique enough to build some sort of signature, then we may be able to use machine learning (ML) techniques to automatically learn malware behaviors and identify them in new variants. In the following sections, we will describe an approach to use existing, well known ML algorithms for exactly this purpose. We will also apply this approach to malware on Android/ARM and Linux/x86 systems, demonstrating its feasibility and evaluating several ML algorithms.

6.4 Experimental Setup

Can simple performance metrics be used to identify malware? To answer this question we conduct several feasibility studies. In each, we collect performance counter data on malware and train a set of classifiers to detect malicious behavior. In addition to data from malware programs, we collect data from non-malware programs (Figure 6.3). Here we describe our program sets, provide details of our data collection infrastructure, describe our classifiers, and discuss types and granularity of malware detection.

6.4.1 Malware & Non-Malware Programs Used

In this study we used 503 malware and 210 non-malware programs from both Android ARM and Intel X86 platforms. The full list of programs is available in the dataset website[1]. The malware programs were obtained from three sources. First from the authors of previous work studying Android malware [163], and second from a website[2] that contains a large number of malware. We also obtained two publicly available Linux x86 rootkits [22, 110]. Data from non-malware programs serve two purposes: during training as negative examples, and during testing to determine false positive rates, *i.e.,* the rate of misclassifying non-malware.

For the purposes of this chapter, we use a wide definition of malware. Malware is any part of any application (an Android APK file or rootkit binary) that has been labeled as malware by a security analyst. We use this definition to enable experimentation with a large amount of malware, which is necessary for supervised machine learning.

This definition of malware is, however, imprecise. Much malware comes attached to legitimate code, so users often execute malware alongside their desired applications. As such, an accurate definition would require malware samples that have undergone deep forensic analysis to determine exact portions that result in malicious actions, and to identify inputs or environmental conditions under which the malware actually performs malicious actions.

As researchers designing a problem for supervised machine learning algorithms, this presents a particular challenge: what parts of our "malware" data should be labeled as such for training? Should we label the entirety of the software as malicious while much of our "malicious" training data could actually be mostly benign? The only other option is to laboriously pick out the good threads or program portions from the bad. This latter option, however, is neither scalable nor practical and to the best of our knowledge not available

[1]`http://castl.cs.columbia.edu/colmalset`

[2]`http://contagiominidump.blogspot.com/`

even in datasets from commercial vendors [45].

While our definition of malware is imprecise, it is practical. However, it makes our classification task more difficult since our classifiers see both malicious behaviors and legitimate behaviors with “malware” labels during training. With more accurate labels we would likely see lower false positive rates and higher malware identification rates. In other words, our experimental framework is conservative and one would expect better results in practice. Our experiments are only designed only to demonstrate feasibility.

6.4.2 Data Collection

Most existing processors have performance counters. They can be configured to count a variety of events such as cycles, instructions, cache misses, etc. Typically they are used to assist in software performance optimization. In this chapter, however, we want to collect time-series data for multiple events simultaneously. For instance, we might want to know when an application simultaneously has high load density, no instruction cache misses and perfect branch prediction. This condition could indicate that a cache side-channel attacker is running.

Unfortunately, existing tools for performance counter collection are not appropriate for this task. Typical tools configure counters for independent sampling; they set each counter to overflow and interrupt once every N events and sample the program’s instruction pointer during each interrupt. Instead, we have written a Linux kernel module that interrupts once every N cycles and samples *all* of the event counters along with the process identifier of the currently executing program. Using this tool we collect multidimensional time-series traces of applications like those shown in Figure 6.2.

Our data collection tool is implemented on two platforms. For x86 workloads, we run Linux 2.6.32 on an 8 core (across two sockets) Intel Xeon X5550 PC with TurboBoost up to 2.67GHz and 24GB of memory. These processors are based on Intel’s Nehalem design, which implement four configurable performance counters, so our Intel x86 data is 4-dimensional. For ARM workloads, we run Android 4.1.1-1 which is based on Linux 3.2. We use a distribution of Android from Linaro that runs on Texas Instrument’s PandaBoard, a demonstration board for their OMAP4460 processor with dual ARM Cortex-A9 cores.

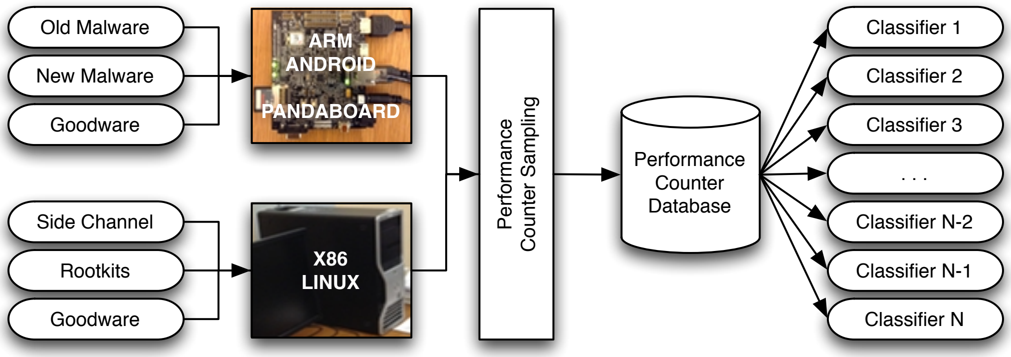


Figure 6.3: Our workflow for malware experiments.

ARM architectures of this generation have six configurable performance counters, so ARM data is 6-dimensional.

Performance As a result of interrupting once every N cycles, the programs we are monitoring suffer some performance degradation. To determine the severity, we run SPEC on both platforms while collecting performance data once every 10,000 cycles. On Intel, this results in a 24.7% geomean slowdown; on ARM, a 24.4% geomean slowdown. To mitigate overheads, for the data collection in the our other experiments we use sampling periods of 50,000 and 25,000 cycles for Intel and ARM respectively. At these periods, the slowdowns largely go away.

Events On ARM, we configure the counters for the following six events as described by the ARM reference manual [8]:

0x06 Memory-reading instruction architecturally executed. This counter increments for every instruction that explicitly read data, including SWP. This counter does not increment for a conditional instruction that fails its condition code check.

0x07 Memory-writing instruction architecturally executed. The counter increments for every instruction that explicitly wrote data, including SWP. This counter does not increment for a Store-Exclusive instruction that fails, or for a conditional instruction that fails its condition code check.

0x0C Software change of PC, except by an exception, architecturally executed. This counter does not increment for a conditional instruction that fails its condition code check.

0x0D Immediate branch architecturally executed:

- B{L} <label>
- BLX <label>
- CB{N}Z <Rn>,<label>
- HB{L} #HandlerId (ThumbEE state only)
- HB{L}P #<imm>, #HandlerId (ThumbEE state only).
- This counter counts for all immediate branch instructions that are architecturally executed, including conditional instructions that fail their condition code check.

0x0F Unaligned access architecturally executed. This counts each instruction that is an access to an unaligned address. That is, the instruction either triggered an unaligned fault, or would have done so if the CPSR.A bit had been 1. This counter does not increment for a conditional instruction that fails its condition code check.

0x12 Branch or other change in program flow that could have been predicted by the branch prediction resources of the processor.

On Intel, we configure the counters for the following four events as described by the Intel reference manual [76]:

L1D_CACHE_LD.E_STATE Counts L1 data cache read requests where the cache line to be loaded is in the E (exclusive) state.

L2_RQSTS.LOADS Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.

UOPS_EXECUTED.PORT0 and .PORT1 Counts number of Uops executed that were issued on port 0/1. Port 0 handles integer arithmetic, SIMD and FP add Uops. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide Uops.

BR_INST_EXEC.ANY Counts all near executed branches (not necessarily retired). This includes only instructions and not micro- op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.

Bias Mitigation We aim to mimic real-world deployment conditions as much as possible when collecting data. There are a variety of factors that could affect our results: (1) Contamination – malware does its best to infect machines and be persistent, possibly influencing subsequent data captures. We control for this by wiping and restoring all non-volatile storage in between data captures for different malware families and, more importantly, between data collection runs of the training and testing set. (2) Environmental noise and input bias: these two factors cannot be controlled in deployment conditions, so in order to make our problem both more difficult and realistic, we do not control for them. (3) Network connectivity: some malware requires an internet connection, so our test systems were connected over Ethernet and were not firewalled or controlled in any way, as they would be in the wild. (4) User bias: We had three different users collect data in arbitrary order for the training and testing runs to mitigate systematic biases in interacting with applications. (5) Ensuring successful malware deployment: We cannot say with certainty if malware actually worked during a run. While the consequences were clear for some malware such as adware, for some malware we observed unexplainable behaviors, such as the system crashing. It is unknown to us whether these bizarre behaviors were intended or not (there are no specification documents for malware), so all data collected was included, possibly polluting our training and testing data, again likely making our classification task more difficult.

6.4.3 Machine Learning Methods

In machine learning, classifiers are able to examine data items to determine to which of N groups (classes) each item belongs. Often, classification algorithms will produce a vector of probabilities which represent the likelihoods of the data item belonging to each class. In the case of malware detection, we can simply define two classes: malware and non-malware. As a result, the output from each of our classifiers will be two probabilities representing the likelihood of the data item being malware.

Features Our data collection produces multidimensional time series data. Each sample is a vector made up of event counts at the time of sampling. In addition to that, we can also aggregate multiple samples, and then use the aggregate to build feature vectors. Aggregation can even out noise to produce better trained classifiers or dissipate key signals depending on the level of aggregation and the program behavior. In this chapter, we experiment with a number of different feature vectors: (1) raw samples (2) aggregations all the samples between context swaps using averages or sums, (3) aggregations between context swaps (previous option) with a new dimension that includes the number of samples aggregated in a scheduling quanta, (4) histograms in intervals of execution. This last one, histograms, breaks up the samples into intervals of fixed size (32 or 128 samples) and computes discrete histograms (with 8 or 16 bins) for each counter. It then concatenates the histograms to create large feature vectors (192 or 768 features on ARM).

Classifiers There are a large number of classifiers we could use. Classifiers broadly break down into two classes: linear and nonlinear. Linear algorithms attempt to separate n-dimensional data points by a hyperplane – points on one side of the plane are of class X and points on the other side of class Y. Nonlinear classifiers, however, have no such restrictions; any operation to derive a classification can be applied. Unfortunately, this means that the amount of computation to classify a data point can be very high. In choosing classifiers to implement for this chapter, we choose to focus on nonlinear algorithms as we did not expect our data to be linearly separable. Here we briefly describe the algorithms we implement:

KNN In k-Nearest Neighbors (KNN), the classifier is trained by inserting the training data points along with their labels into a spatial data structure like a kd-tree. In order to classify a data point, that point's k nearest neighbors (in Euclidean space) are found using the spatial data structure. The probability that the data point is of each class is determined by how many of its neighbors are of that class and their Euclidean distance. We train and test KNN classifiers using k = 5, 10, and 25.

Decision Trees Another way to classify data points is to use a decision tree. This tree is built by recursively splitting training data into groups on a particular dimension. The dimension and split points are chosen to minimize the variance in training data within each group. These decisions can also integrate some randomness, decreasing the quality of the tree but helping to prevent over training. After some minimum variance is met, a maximum depth is reached, or minimum number of data points remaining in the branch is hit, a branch terminates, storing in the node the mix of labels in its group. To classify a new data point, the decision tree is traversed to find the new point's group (leaf node) and return the stored mix. We train and test decision tree classifiers with minimum number of training data points per branch of 5, 10, and 25.

Random Forests One way to increase the accuracy of a classifier is to use multiple different classifiers and combine their results. In a random forest, several (or many) decision trees are built using some randomness. When classifying a new data point, the results of all trees in the forest are weighted equally. We train and test random forests with 5, 10, 25, and 50 decision trees.

ANN Finally we attempt classification with Artificial Neural Networks (ANNs). In our neural nets, we define one input neuron for each dimension and two output nodes: one for the probability that malware is running, and one for the probability that non-malware is running. We train and test ANNs using RProp or Quickprop back-propagation, 3 or 5 layers, and 5 or 10 neurons per layer.

For implementation, we use KNN, Decision Trees, and Random Forests from the Waffles ML library[3]. For our ANNs, we use the FANN library[4].

Classifying Time-Series Data The ML algorithms listed above are capable of classifying only fixed dimension vectors and thus cannot themselves classify time-series data. To classify our multidimensional time-series data (a program thread or multiple threads), we train the classifiers on the vectors from each time step in our data, discarding all notions

[3]waffles.sourceforge.net 2012-08-31

[4]fann.sourceforge.net FANN-2.2.0

or ordering and time. During testing, we run the classifier on each vector in the time series, resulting in a unidimensional time-series of probabilities that malware is running. To compute the probability that an entire time-series or multiple time-series were generated by malware, we simply average the probabilities of all points in time.

6.4.4 Training and Testing Data

As mentioned before many production malware detectors build blacklists using static malware signatures. As a result, they can only detect malware that the AV vendor has already discovered and cataloged. Minor variations thereof – which are relatively easy for attackers to produce – cannot be detected in the wild using existing signatures. If we wanted to, we could design a hardware detector that works exactly as the software signature AV. We would evaluate the feasibility of this by running the same malware multiple times under different conditions to produce the training and testing data. But in this work we want to design a more robust malware detector that in addition to detecting known malware, will also detect new variants of known malware. In order evaluate this functionality, we train a classifier on data from one set of programs – non-malware and variants of malware in a family. We then test the classifier's accuracy on different variants of malware in the same family (and also on non-malware programs). To mitigate bias, the data for training and testing are collected in separate runs without knowledge of whether the data is to be used for testing or training. The data is also collected by different users.

6.4.5 Classification Granularity

Our data collection can procure performance counter data every 25,000 or 50,000 cycles with little slowdown. So in theory we can classify malware at the granularity of each sample. However, due to large number of small variations in programs we should expect a large number of false positives. We have indeed found this to be the case, and in fact, we obtained high false positives even at a coarser granularity of every operating system context swap. As such, in this chapter, we present classification results for malware at two even coarser granularities: thread and application group. In the thread based classification, each thread is classified as malware (or non-malware) by aggregating the classification probabilities for all data points in that thread. In application group level classification, we classify Android Java packages and package families as malware. This approach requires our classifier to determine if, for example, "com.google.chrome" is malicious or not and allows the classifier to use samples from any thread executing that code.

6.5 Detecting Android Malware

With the rise of Android has come the rise of Android malware. Android has the concept of permissions; during each package install, software must ask the user permission to do certain things, *i.e.,* read GPS location, access the network, et cetera. However this permissions-based approach often fails because users typically provide permissions indiscriminately or can be tricked into giving permissions by the application. For instance, a fake application packaged like Skype can trick the user into giving permissions to access the camera and microphone. In fact, several Android malware applications mask themselves as legitimate software, and it is not uncommon for malware writers to steal existing software and repackage it with additional, malicious software.

6.5.1 Experimental Design

The Android malware data sets are divided up into families of variants. In families with only one variant, we use the same malware but different executions of it for training and testing. For families with more than one variant, we statically divide them up, using $\frac{1}{3}$ for training and the rest for testing. The names of each family and the number of installers (APKs) for each can be found in our results, Table 6.2. In total our data set includes nearly 368M performance counters samples of malware and non-malware.

Classifier Parameters The classification algorithms outlined in Section 6.4 can be parameterized in different ways. For instance, for k-Nearest Neighbors, k is a parameter. We search a large space of classifiers, varying many parameters. In order to determine the best set of parameters, we want to choose the classifier that identifies the most malware correctly. However, as we make the classifier more sensitive, we find more malware but also identify some legitimate software as malware. In order to determine which classifier to use, we find the one that performs best (on the training data) for a given false positive percentage. As a result, the results we present are not necessarily monotonically increasing.

Re-training Optimization Since malware applications are known to include both malicious and benign code, we use an optimization to select data points for training that are more likely to be from the malicious part of an malware package. We first train our classifier on all data points. We then run all of our training data through this classifier and sort the data based on the classifier's score, *i.e.,* we calculate the probability of data being malware as called by the malware classifier. We then use only the most "malware-like" data in our training set to re-train the classifier, which we then use in evaluation. The intuition behind this technique is that the non-malicious parts of our training data are likely to look a lot like non-malware to the classifier, so we use our initial classifier to filter out those data. In many cases, this retraining allows us to retrain with a smaller amount of data while achieving comparable accuracy (and speeding up training and testing.) In the case of decision trees, we find that this technique significantly improves results. Further, it creates relatively small decision trees, so the computational requirements of classifying each sample is orders of magnitude lower than some of the other methods.

Next we report results on detecting malware at the granularity of threads and at the application level.

6.5.2 Malware Thread Detection Results

Testing The classification metric we use is the percentage of threads correctly classified. For instance if the malware application has T threads, our classifier, in the ideal case, will flag only those subset of threads that perform malicious actions. For non-malware, ideally all threads should be flagged as benign. As mentioned before, the testing data samples are obtained from a separate run from training and under different input and environmental conditions. We also use different non-malware applications in testing than in training to ensure that we do not build a *de facto* white- or blacklist of applications.

Training We ensure that an equal number of samples from malware and non-malware are used for training. Strictly speaking this is unnecessary but we did it to prevent our classifier results from being biased by the volume of samples from the two categories. The

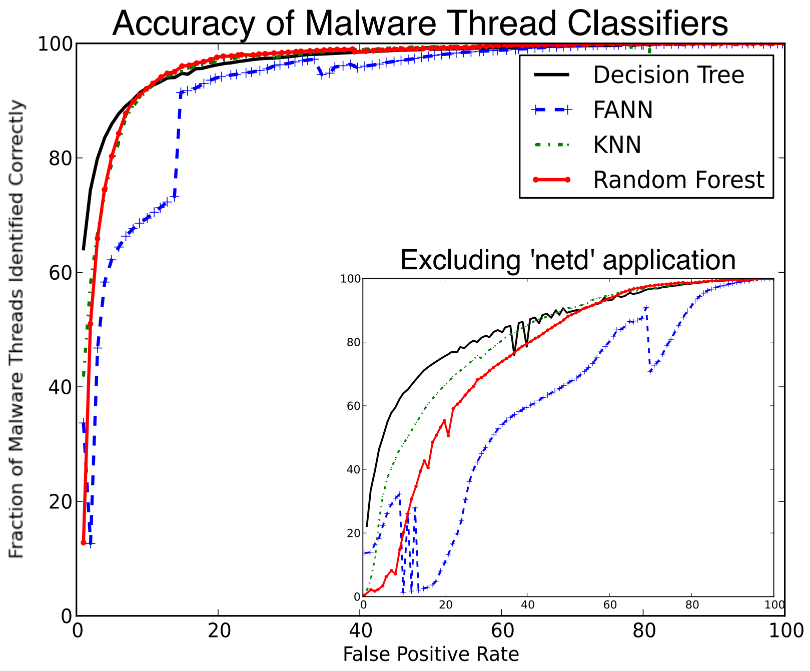


Figure 6.4: The accuracy of binary classifiers in determining whether or not each running thread is malware.

samples are chosen without any relation to the number of threads to mitigate classification bias due to thread selection.

Results Figure 6.4 shows malware detection by thread in a form similar to a typical ROC curve. As expected, if we allow some false positives, the classifiers find more malware. These results indicate that performance counter data can, with simple analysis, be used to detect malware with relatively good accuracy. To verify the statistical significance of our findings, we can show that the malware population is significantly different from the non-malware population. These populations are constructed from the malware probabilities of all the threads in each set of threads. Running Student's T-Test [144] on these two populations, we find a p-value of 0.0000, demonstrating that our results are statistically significant.

Further analysis of results shows that a single application makes up the majority of non-malware during the testing phase. This application is an Android system application called "netd" and is responsible for dynamically reconfiguring the system's network stack. As

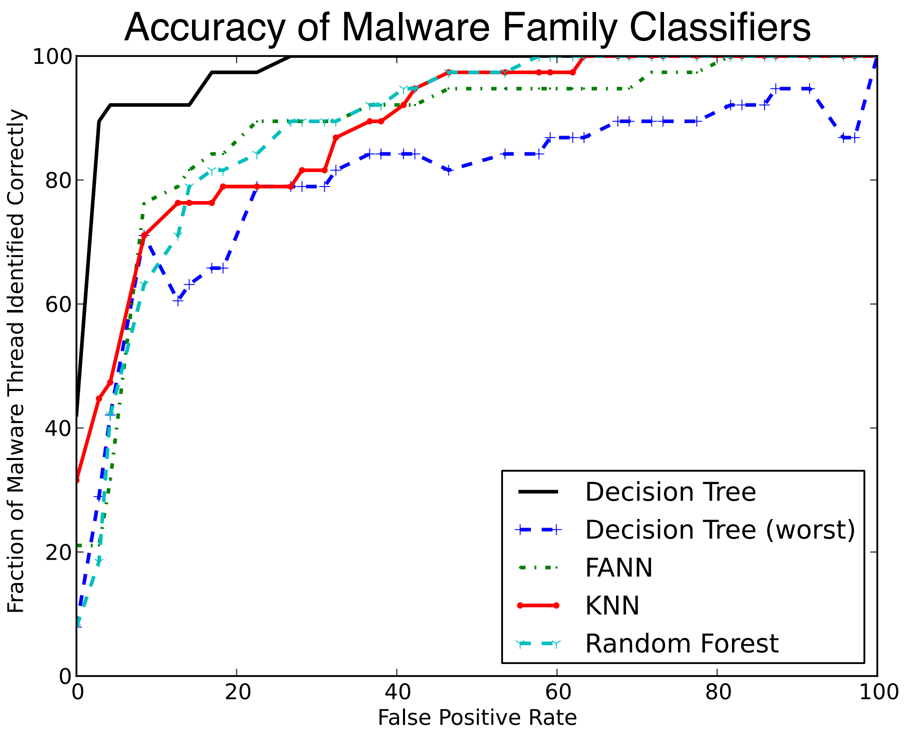


Figure 6.5: The accuracy of binary classifiers in determining whether families of malware and normal Android packages are malware.

such, it runs often, and our classifiers are excellent at correctly predicting this application as non-malware. If we remove this application from our testing data, we obtain the results inlaid in Figure 4. While they are not as good, they remain positive.

We further break down our results by malware family in Table 6.2. This table shows the number of APKs we were able to obtain for each family along with the number of threads observed. It also shows the number of threads that our classifier correctly identified while maintaining a 10% or better false positive rate. We find a range of results, depending on the family.

6.5.3 Malware Package Detection Results

Testing For application/package-based malware detection, our classifiers use samples from all the threads belonging to a particular software. For instance, all of the samples collected from the testing set of Anserverbot are used to determine whether or not that set of software

Table 6.2: Malware Families for Training and Testing

Malware Family	APKs	Training Threads	Testing Threads	Threads Flagged	Rate
Tapsnake	1	31	3	3	100%
Zitmo	1	5	1	1	100%
Loozfon-android	1	25	7	7	100%
Android.Steek	3	9	9	9	100%
Android.Trojan. Qicsomos	1	12	12	12	100%
CruseWin	1	2	4	4	100%
Jifake	1	7	5	5	100%
AnserverBot	187	9716	11904	11505	96.6%
Gone60	9	33	67	59	88.1%
YZHC	1	9	8	7	87.5%
FakePlayer	6	7	15	13	86.7%
LoveTrap	1	5	7	6	85.7%
Bgserv	9	119	177	151	85.3%
KMIN	40	43	30	25	83.3%
DroidDreamLight	46	181	101	83	82.2%
HippoSMS	4	127	28	23	82.1%
Dropdialerab	1	18*	16*	13	81.3%
Zsone	12	44	78	63	80.8%
Endofday	1	11	10	8	80.0%
AngryBirds-LeNa.C	1	40*	24*	19	79.2%
jSMSHider	16	101	89	70	78.7%
Plankton	25	231	551	432	78.4%
PJAPPS	16	124	174	136	78.2%
Android.Sumzand	1	8	9	7	77.8%
RogueSPPush	9	236	237	184	77.6%
FakeNetflix	1	27	8	6	75.0%
GEINIMI	28	189	203	154	75.9%
SndApps	10	110	77	56	72.7%
GoldDream	47	1160	237	169	71.3%
CoinPirate	1	8	10	7	70.0%
BASEBRIDGE	1	14*	72	46	63.8%
DougaLeaker.A	6	12*	35*	22	62.9%
NewZitmo	1	5	8	5	62.5%
BeanBot	8	122	93	56	60.2%
GGTracker	1	16	15	9	60.0%
FakeAngry	1	7	10	5	50.0%
DogWars	1	14	8	2	25.0%

* Indicates that data collectors noticed little activity upon launching one or more of the malware APKs, so we are less confident that the payload was successfully achieved.

Table 6.3: Malicious Package Detection Results: Raw Scores

Score	Malware Family	Score	Malware Family	Score	Malware Family
0.67	YZHC	0.61	CruseWin	0.58	NewZitmo
0.66	Tapsnake	0.61	BASEBRIDGE	0.58	DogWars
0.65	Android.Sumzand	0.61	Bgserv	0.57	GEINIMI
0.65	PJAPPS	0.61	DougaLeaker.A	0.56	FakePlayer
0.64	Loozfon-android	0.61	jSMSHider	0.56	AngryBirds-LeNa.C
0.63	SndApps	0.61	FakeAngry	0.55	Android.Trojan.Qicsomos
0.63	GGTracker	0.61	Jifake	0.53	GoldDream
0.62	Gone60	0.61	RogueSPPush	0.53	RogueLemon
0.62	FakeNetflix	0.60	Android.Steek	0.53	AnserverBot
0.62	Zsone	0.60	Dropdialerab	0.49	Plankton
0.62	CoinPirate	0.60	HippoSMS	0.49	BeanBot
0.62	Zitmo	0.60	Endofday	0.47	LoveTrap
0.61	DroidDreamLight	0.59	KMIN	0.59	Average

Score	Goodware
0.55	appinventor.ai_todoprogramar.
	HappyWheelsUSA
0.53	com.android.keychain
0.53	com.pandora.android
0.51	com.bestcoolfungames.antsmasher
	....
0.38	com.twitter.android
0.38	com.android.packageinstaller
0.37	com.android.inputmethod.latin
0.36	android.process.media
0.44	Average

is malware.

Training In the previous experiment on detecting malware granularity by threads, we used an equal number of samples for both malware and non-malware, but did not normalize the number of samples by application or malware family. In this study, however, in addition to using an equal number of samples for non-malware and malware, we use an equal number of samples from each malware family and an equal number of samples from each non-malware application. This ensures that during training our classifiers see data from any application that ran for a non-trivial amount of time and they are not biased by application run times during the training phase. Since we want to have equal number of samples, we leave out short-running applications and malware families that produce fewer than 1,000 samples.

Results The results of our package classifiers are found in Table 6.3 and Figure 6.5. The results are equally positive by application as they are for threads. As in the last experiment (thread results), we ran a large number of classifiers with different parameters and selected the best parameter for each false positive rate based on the accuracy of the classifier on the training data (these were 100s of different classifiers). However, unlike the last study, we found that our decision tree classifiers did near-perfectly on all the training data so we could not pick one best parameter configuration. In Figure 6.5 we show the best and worst accuracies we obtained with different parameters for the decision trees which performed near-perfectly on the testing data. Future work should consider a methodology for selecting classifier parameters in such cases of ties. The table shows raw classifier scores for our malware and some non-malware, both sorted by score. The particular classifier results showcased here aggregate raw decision tree scores from all samples collected from each malware family and non-malware package. We see that on average our malware scores are higher for malware than non-malware. There is, however, some overlap, creating some false positives.

6.5.4 Conclusions on Android Malware

In our experiments we are testing on a different set of variants from those we train on, showing that our classifiers would likely detect new malware variants in the field that

Table 6.4: AUC below 10% False Positive Rates

Classifier	Thread Detection	Package Detection
Decision Tree	82.3	83.1
KNN	73.3	50.0
Random Forest	68.9	35.7
FANN	53.3	38.4

security investigators had not yet seen. Table 6.4 shows the area under the curve for both schemes with 10% false positives. This is a capability that static signature-based virus scanners lack, so there is little basis for comparison. We also showed that our results are consistently positive for two different detection granularities (and thus metrics) increasing our confidence in our malware detection scheme.

Are these results as good as they can be? We are unable to answer this question. The reason is that malware often includes both malicious and non-malicious code, but we do not attempt to separate them. As a result, we label all the threads in malware APKs as malicious in our testing set. But what if only half the threads are responsible for malicious behavior whereas the other half are legitimate code which was not present in our training data? Were this the case, it could well be that we are perfectly detecting all the malicious threads.

Nonetheless, many of our results are quite promising. For instance, after training on data from only five of our YZHC variants, the remaining variants are given significantly higher malware scores than our unseen non-malware. Similarly, after training on only $\frac{1}{3}$ of AnserverBot's variants, threads from the remaining variants are tagged as malware far more often than are non-malware. With further refinement in terms of labeling data and better machine learning methods, we expect that accuracy could be improved significantly.

6.6 Detecting Linux Rootkits

Rootkits are malicious software that attackers install to evade detection and maximize their period of access on compromised systems. Once installed, rootkits hide their presence, typically by modifying portions of the operating systems to obscure specific processes, network ports, files, directories and session log-on traces. Although there exist open-source tools like *chkrootkit*[5] and *rkhunter*[6] to detect rootkits, their use of known signatures makes it easy for rootkits to evade detection by varying their behaviors. Furthermore, since these tools work on the same software level as the rootkits, they can be subverted.

6.6.1 Experimental Design

In this case study, we examine the feasibility of rootkit detection with performance data. We examine the two publicly available Linux rootkits which give an attacker the ability to hide log-on session traces, network ports, processes, files and directories. The Average Coder Rootkit works as a loadable kernel module that hides traces via hooking the kernel file system function calls [110]. The Jynx2 Rootkit functions as a shared library and is installed by configuring the LDPRELOAD environment variable to reference this rootkit [22].

To exercise these rootkits, we run the "ps", "ls", "who", and "netstat" Linux commands and monitor their execution. The Average Coder rootkit is used to hide processes, user logins and network connections whereas the Jynx2 rootkit affects "ls" to hide files. To introduce some input bias and collect multiple samples for both training and testing, we run each command with a variety of different arguments. We run half the commands before the rootkit is installed and half after. After data collection, we split the executions up into training and testing sets. Since we do not repeat commands with the same arguments, our training data are input biased *differently* from our testing data, making the learning task both more difficult and more realistic. To increase the variability in our data, we also simulate various user actions like logging in/out, creating files, running programs and initiating network connections. Lastly, to protect against contamination, we wiped our system between installation of the rootkits and collection of "clean" data.

For this case study, we also show the results from an additional classifier: tensor density. This classifier discretizes the vector space into many buckets. Each bucket contains the relative density of classes in the training data set. A data point is classified by finding its bin and returning the stored mix. Although simple, the tensor has $O(1)$ lookup time, so

[5]`http://www.chkrootkit.org/`

[6]`http://rkhunter.sourceforge.net/`

the classifier is very time-efficient.

6.6.2 Results

We experimented with five different classifiers, the results of which are presented in Figure 6.6. The "combined" classifier was trained and tested on all of the above programs whereas the other experiments used data from only one of the programs.

Our rootkit identification results are interesting, though not quite as good as the results presented for Android malware in Section 6.5. The reason rootkit identification is extremely difficult is that rootkits do not operate as independent programs. Rather, they dynamically intercept programs' normal control flows. As a result, the data we collect for training is affected only slightly by the presence of rootkits. Given these difficulties, we believe our rootkit detection shows promise but will require more advanced classification schemes and better labeling of the data to identify the precise dynamic sections of execution that are affected.

6.7 Side-Channel Attacks

As a final case study, we look at side-channel attacks. Side-channel attacks are not considered malware. However, they also threaten security, and we find that our methods can be used even to detect these attacks.

In a side-channel attack unintentional leaks from a program are used to infer program secrets. For example, cryptographic keys can be stolen by observing the performance of the branch predictor or of the caches for many microprocessor implementations. Nearly any system is vulnerable to side-channel attacks [40].

In a microarchitectural side-channel attack, a *victim* process is a process that has secrets to protect and an *attacker* process attempts to place itself within the system in such a way that it shares microarchitectural resources with the victim. Then it creates interference with the victim, *e.g.,* thrashes a shared resource constantly so as to learn the activity of the victim process with respect to that shared resource. The interference pattern is then mined to infer secrets. Since the attackers' interference pattern is programmed we intuitively

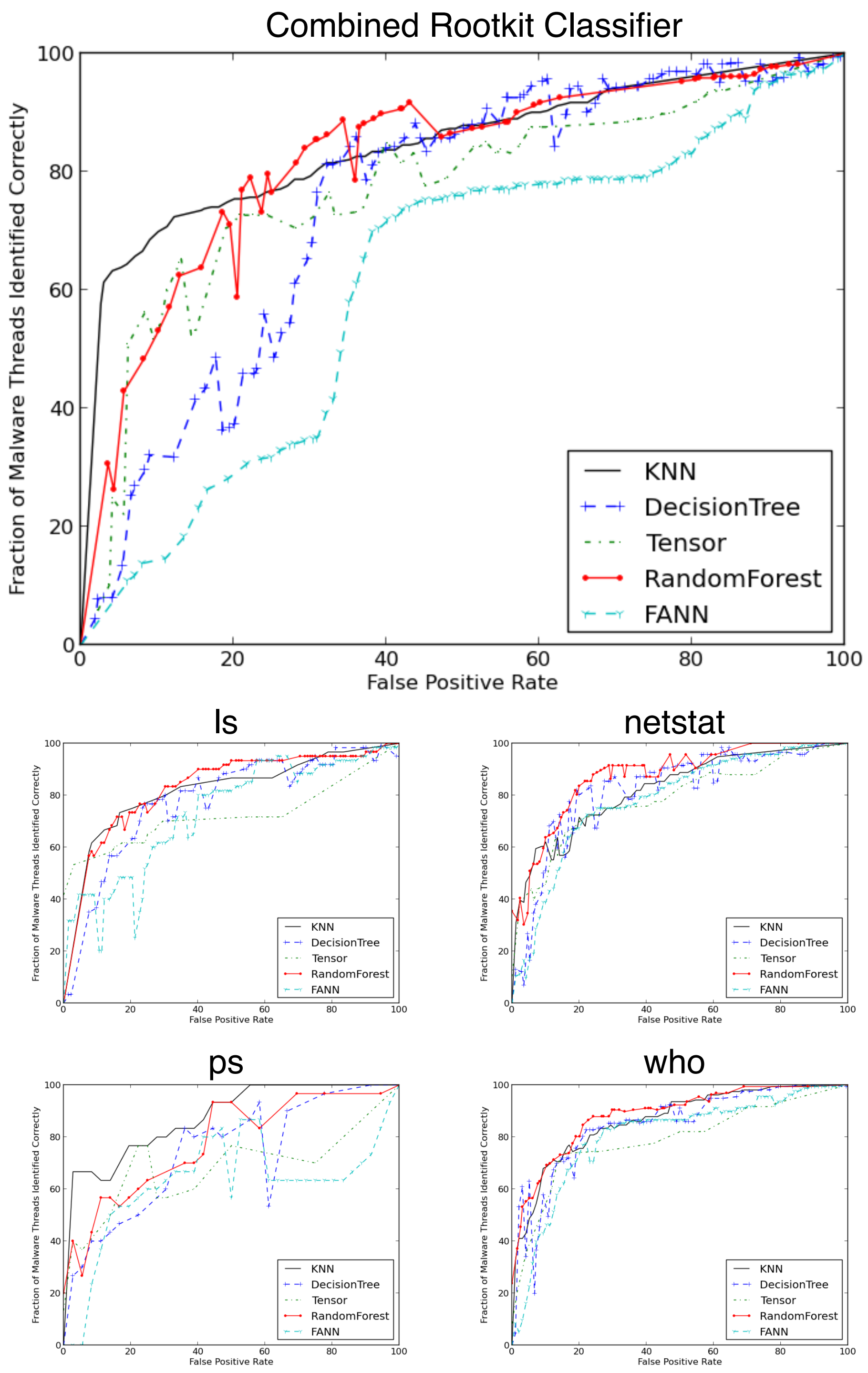


Figure 6.6: Accuracy of rootkit classifiers on several applications in addition to a classifiers trained and test on all of the applications combined.

expect that attacker programs that exploit microarchitectural side-channels should have clear signatures.

Experimental Design To test our intuition we examine one very popular class of side-channel attacks known as a cache side-channel attack. We hypothesize that one particular method for this type of attack - the prime and probe attack method - is a good target for hardware anti-virus. To test our hypothesis, we implement several variants of the standard prime-and-probe technique. In this technique, an attacker program writes to every line in the L1 data cache. The program then scans the cache repeatedly — using a pattern chosen at compile time — reading every line. Whenever a miss occurs, it means there was a conflict miss caused by the victim process sharing the cache. The result of a successful prime-and-probe attack is data about the cache lines used by the victim process over time. Using OpenSSL as the victim process, we compare cache side-channel attack processes against a wide array of benign processes. These benign programs include SPEC2006 int, SPEC2006 fp, PARSEC, web browsers, games, graphics editors and other common desktop applications, as well as generic system-level processes.

Results We train our machine learning algorithms on one third of our total data: 3872 normal program threads and 12 attack threads. We then test our classifiers on the other $\frac{2}{3}$ of the data. Our results in this case are perfect. We catch 100% of the attackers and do not have any false positives on all four classifiers we used. These results demonstrate that cache side-channel attacks are easy to detect with performance counters. We have tested a sub-type of side-channel attacks on one microarchitectural structure but it is likely that other types of microarchitectural side-channel attacks are also detectable. While these initial results are promising further study is necessary to prove this hypothesis.

6.8 Hardware Support

Moving security protection to the hardware level solves several problems and provides some interesting opportunities. First, we can ensure that the security system cannot be disabled by software, even if the kernel is compromised. Second, since the security system runs beneath the system software, it might be able to protect against kernel exploits and

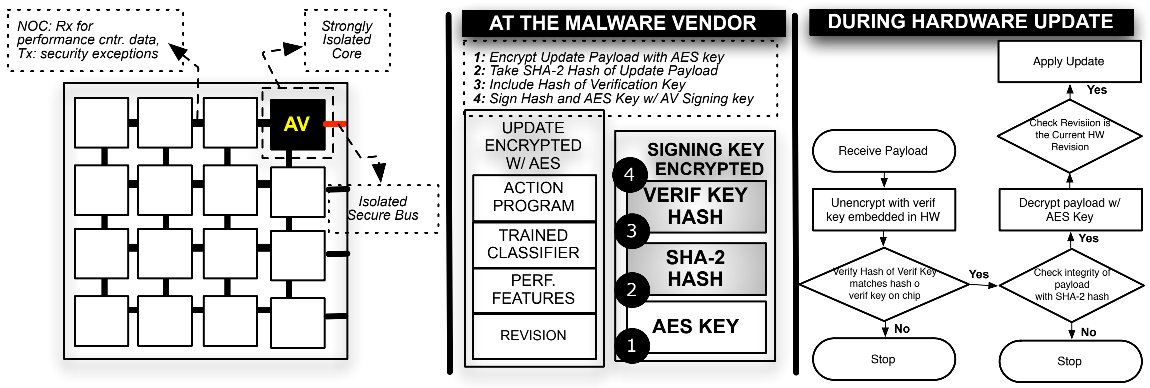

NOC: Rx for performance cntr. data, Tx: security exceptions
Strongly Isolated Core
AV
Isolated Secure Bus
AT THE MALWARE VENDOR
1: Encrypt Update Payload with AES key
2: Take SHA-2 Hash of Update Payload
3: Include Hash of Verification Key
4: Sign Hash and AES Key w/ AV Signing key
UPDATE ENCRYPTED W/ AES
ACTION PROGRAM
TRAINED CLASSIFIER
PERF. FEATURES
REVISION
SIGNING KEY ENCRYPTED
4
VERIF KEY HASH
3
SHA-2 HASH
2
AES KEY
1
DURING HARDWARE UPDATE
Receive Payload
Unencrypt with verif key embedded in HW
Verify Hash of Verif Key matches hash o verif key on chip
Yes
No
Stop
Check integrity of payload with SHA-2 hash
Yes
No
Stop
Decrypt payload w/ AES Key
Check Revisiion is the Current HW Revision
Yes
Apply Update

Figure 6.7: Hardware Architecture for AV Execution and Update Methods for Performance Counter Based AV

other attacks against hypervisors. Third, since we are modifying the hardware, we can add arbitrary static *and* dynamic monitoring capabilities. This gives the security system unprecedented views into software behavior.

The overall hardware security system that we propose is shown in Figure 6.7. The system has four primary components:

Data Collection We must define what data the security processor can collect and how that data is collected and stored.

Data Analysis The security system must analyze incoming data to determine whether or not malicious behavior is occurring.

Action System If a threat is detected by the system, it must react in some manner. This may involve measures as extreme as shutting down the system or as mild as reporting the threat to a user.

Secure Updates Any security measure must, from time to time, be updated to deal with the latest malware. However, these updates must be secure to ensure that only a trusted authority can update the system.

There are many ways to implement a hardware malware detector. The most flexible solution is to allocate one or more general-purpose cores which allows any classification algorithm to be used for detection. Alternatives include microcontrollers or microcontrollers with special-purpose malware detection units that are located on chip, on-chip/off-chip FPGA, and off-chip ASIC co-processor. These choices represent different trade-offs in terms of flexibility and area- and energy-efficiency that need to be explored in detail in the future. In the rest of the section, however, we focus on the backbone system framework required to realize any of these design choices. As we discuss the system framework, we make recommendations or highlight research advancements needed to enable online malware detection with performance counters.

But first, a note on terminology: irrespective of the design choice *i.e.,* microcontroller, accelerator, big or little cores, on-chip unit or off-chip co-processor, FPGA or ASIC, we refer to the entity hosting the classifier algorithm as the AV engine and the units running the monitored programs as targets.

6.8.1 System Architecture

The system architecture should allow the AV engine: (1) to run independently of any operating system or the hypervisor, and at the highest privilege level in the system. This is to enable continuous monitoring of software at all levels in the stack (2) to enable access to physical memory to store classifier data and (3) to provide strong memory and execution isolation for itself. Isolation ensures that the AV engine is not susceptible to denial-of-service attacks due to resource provisioning (*e.g.,* memory under- or over-flow), or resource contention (*e.g.,* stalling indefinitely due to excessive congestion on the network-on-chip).

Some of these features already exist in processor architectures today. For instance, AMD processors allow a core to carve out a region of the physical memory and lock down that physical memory region from access by other cores [10]. Similarly, some architectures support off-chip coprocessors to have dedicated and isolated access to physical memory through IOMMUs. These features must be extended with mechanisms that guarantee starvation-freedom in shared resources such as the memory controller and in the network-on-chip (or buses in the case of an off-chip AV) to ensure robust communication between the AV engine and the targets.

Recommendation #1 Provide strong isolation mechanisms to enable anti-virus software to execute without interference.

6.8.2 Data Collection

From the perspective of implementing an A/V engine in hardware, no additional information beyond performance information and thread ID is necessary for thread-based classification. For application-level classification, hardware will need application level identifiers associated with each thread. Thread IDs can already be obtained by hardware.

The AV engine receives performance counter information periodically from the targets. In our experiments, the data from the performance counters is fetched once every 25,000 cycles. This translates to a bandwidth requirement of approximately a few hundred KB/s per target. If the number of active targets (which is at most cores-times-simultaneous-threads many) is not too large like in today's systems, we can design off-chip AV engines using simple serial protocols (such as I2C) with round-robin collection of data from targets. However, as the number of cores increases, on-chip solutions will become more relevant.

Performance data can be either pulled or pushed from the targets. In the pull model – the model used in our experiments – the targets are interrupted during execution to read their performance counters which impacts performance (roughly 5% empirically). If future hardware support allows performance counters to be queried without interruption, these overheads can be reduced to effectively zero. Another modification that would simplify the design of the AV engine would be to set up the counters to push the data periodically to the AV engine.

The amount of storage required to store the ML data varies greatly depending on the type of classifier used for analysis. For the KNN algorithm, the data storage was roughly 50 MB for binary classification. On the other hand, other analyses needed only about 2.5 MB. Given the variability in storage size and the amount needed, it appears that AV engines will most certainly need mechanisms to access physical memory for retrieving stored signatures.

Recommendation #2 Investigate both on-chip and off-chip solutions for the AV implementations.

Recommendation #3 Allow performance counters to be read without interrupting the executing process.

Recommendation #4 Ensure that the AV engine can access physical memory safely.

6.8.3 Data Analysis

A wide variety of classifiers can be implemented for data analysis. In this chapter we experiment with four well-known classifiers to estimate their potential for malware identification. Most likely, advances in machine learning algorithms and implementations will enable better classification in the future. To allow for this flexibility it appears that general purpose cores are preferable to custom accelerators for the AV engine. However, the AV engine may present domain-specific opportunities for instruction customization, such as special types of memory instructions or microarchitectural innovations in terms of memory prefetchers.

A classification scheme is at best as good as the discerning power of its features. We show that current performance counters offer a good number of features that lead to good classification of malware. However, it is likely that the accuracy can be improved further if we included more features. Thus, we add our voice to the growing number of performance researchers requesting more performance counter data in commercial implementations. Specifically, from the point of view of our malware detection techniques, information regarding instruction mixes and basic block profiles for regions would be very helpful. These inputs can inform the analysis of working-set changes.

Recommendation #5 Investigate domain-specific optimizations for the AV engine.

Recommendation #6 Increase performance counter coverage and the number of counters available.

6.8.4 Action System

Many security policies can be implemented by the AV engine. Some viable security policies are:

• *Using the AV engine as a first-stage malware predictor.* When the AV engine suspects a program to be malicious it can run more sophisticated behavioral analysis on the program. Hardware analysis happens ‘at speed’ and is orders of magnitude faster than behavioral analysis used by malware analysts to create signatures. Such pre-filtering can avoid costly behavioral processing for non-malware programs.

• *Migrating sensitive computation.* In multi-tenant settings such as public clouds, when the

AV engine suspects that an active thread on the system is being attacked (say through a side-channel) then the AV engine can move the sensitive computation. Of course, in some scenarios it may be acceptable for the AV system to simply kill a suspect process.

• *Using the AV engine for forensics.* Logging data for forensics is expensive as it often involves logging all interactions between the suspect process and the environment. To mitigate these overheads, the information necessary for forensics can be logged only when the AV engine suspects that a process is being attacked.

Thus there are a broad spectrum of actions that can be taken based on AV output. The AV engine must be flexible enough to implement these security policies. Conceptually, this means that the AV engine should be able to interrupt computation on any given core and run the policy payload on that machine. This calls for the AV engine to be able to issue a non-maskable inter-processor interrupt. Optionally, the AV engine can communicate to the OS or supervisory software that it has detected a suspect process so that the system can start migrating other co-resident sensitive computation.

Recommendation #7 The AV engine should be flexible enough to enforce a wide range of security policies.

Recommendation #8 Create mechanisms to allow the AV engine to run in the highest privilege mode.

6.8.5 Secure Updates

The AV engine needs to be updated with new malware signatures as they become available or when new classification techniques are discovered. The AV update should be constructed in a way to prevent attackers from compromising the AV. For instance, a malicious user should not be able to mute the AV or subvert the AV system to create a persistent, high-privilege rootkit.

We envision that each update will contain one or more classifiers, an action program that specifies security policies, a configuration file that determines which performance features are to used with what classifiers, and an update revision number. This data can be delivered to the AV engine securely using techniques used for software signing but requires a few tweaks to allow it to work in a hardware setting. The process is described in Figure 6.7.

First, we require that the AV engine implements the aforementioned flowchart directly in hardware: this is because we do not want to trust any software, since all software is potentially vulnerable to attacks. Second, we require hardware to maintain a counter that contains the revision number of the last update and is incremented on every update. This is to prevent an attacker from rolling back the AV system, which an attacker might do to prevent the system from discovering new malware. The AV engine offers this protection by rejecting updates from any revision that is older than the revision number is the hardware counter. In other words, there are fast-forwards but no rewinds.

Recommendation #9 Provide support in the AV engine for secure updates.

6.9 Conclusions

In this chapter we investigate if malware can be detected in hardware using data available through existing performance counters. If possible, it would be a significant advance in the area of malware detection and analysis, enabling malware detection with very low overheads. Further, it would allow us to build malware detectors which are invisible to the system, in the hardware beneath the operating system.

The intuition that drove us to ask this question was the observation that programs appear to be unique in terms of their time-series behavior, while variants of the same programs do similar things. Our results indicate that this intuition is true. We can often detect small changes to running programs (rootkit detection) or be somewhat insensitive to variations (malware detection) depending on how we train our classifier.

We demonstrate the feasibility of our detection methods and highlight the increased security from leveraging hardware, but more research is necessary. First, our detector accuracy can be improved. This will involve further research into classification algorithms and ways to label malware data more accurately. Second, our classifiers are not optimized for hardware implementations. Further hardware/algorithm co-design can increase accuracy and efficiencies.

Despite our results it is not clear if dynamic analysis like ours provides a significant advantage to defenders in the malware arms race. While we are able to detect some variants, could virus writers simply continue permuting their malware until it evades our detector? Would this level of change to the virus require some human intervention, making the task more difficult? We suspect that our techniques increases difficulty for virus writers. This is because the virus writer now needs to take into account a wide range of microarchitectural and environmental diversity to evade detection. This is likely difficult, thus the bar for repeatable exploitations is likely to be higher. However, this topic merits further study.

Part III

Conclusions

Chapter 7

Recommendations for Data Collection and Processing

Since this dissertation is about measurement and analysis, we have implemented data collection and processing infrastructures for each of the described projects. This implementation has been instructive in terms of providing guidance and lessons for how these systems should be implemented in the future. In this chapter, we reflect on some of the implementation difficulties and providing some recommendations for future systems.

7.1 Software for Data Collection, Storage, and Analysis

Each of the projects described in this dissertation involved the collection of data (through measurement hardware or simulation software), storing this data, and analyzing the data with custom software. In this section, we review the technical requirements of each project from a software perspective and make some recommendations for scientific database software based on our experience.

7.1.1 Requirements

While the data storage and processing requirements of this dissertation's projects may sound like exactly the job of a database, no existing database system had both the capabilities ease-of-use we required. Instead, for each project a rather large amount of infrastructure

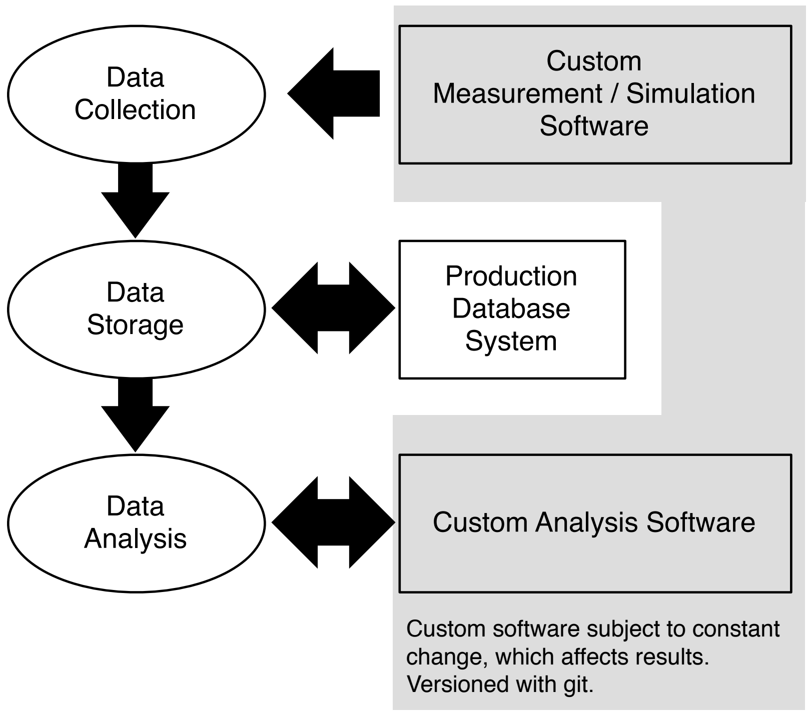

Data Collection
Custom Measurement / Simulation Software
Data Storage
Production Database System
Data Analysis
Custom Analysis Software
Custom software subject to constant change, which affects results. Versioned with git.

Figure 7.1: Interactions between various softwares and steps in all four projects.

was developed to fulfill these needs and non-trivial amounts of data management were conducted manually. For two of the projects a database was used, but only for the storage requirement. Ideally, a database system would have assisted in all three stages.

The interactions for these various pieces are shown in Figure 7.1. There are two big logistical problems we encountered which could be fixed with better software: (1) The two pieces of custom software tend to change often to support new features and fix bugs. The problem is that whenever one changes, it can affect the data which was collected or the analysis of the data. (2) Neither piece of custom software (which could be written in a variety of different languages) is able to nicely integrate with the database software. Instead, they are merely clients of the database software; this works, but often it means that the analysis software simply reads all the necessary data out of the database and does all its processing in its own memory. As a result, much of the analysis software has rather long runtimes (mostly copying data from the database) even though its analysis is very simple.

7.1.2 Recommendations

Instead of a traditional database system, a data processing controller is needed. A user should be able to quickly and easily define versioned software which can produce data, and versioned software which operates on stored data. The system could initiate either analysis or data collection and records results along with version information about the software which produced it. For efficiency, it also needs some way of providing access to data with little overhead and providing some data analysis primitives. Ideally, the system could simply share the memory space of its data with the custom software and provide a library for database operations.

Finally, this system must be very easy to use. Technically, existing SQL databases support all of these features through to use of user-defined functions, other plugin methods, and by attempting to program some of the analyses in SQL. However, the structure and programming of their plugin systems tend to be rather Byzantine. Further, setting up and maintaining schemas for traditional databases adds too much overhead to development; often it is difficult to plan out the necessary data format before hand. Finally, though some of the analyses in this dissertation could have been coded in SQL or a data query language, many were quite complicated and required low-level languages like C++ in order to operate at sufficient speed.

Some of the requirements listed here are similar to those of SciDB [36, 143]. In particular, issues of not overwriting data and provenance are essentially the same. SciDB is also intended for environments where fast, efficient data analysis is key, though their plugin system seems no less complex than is traditional. Also, SciDB requires predefined schemas and does not integrate with source code versioning systems. With some further integration work, however, SciDB could fill the role which these projects required.

7.2 Hardware for Measurement

One of the central focuses of this dissertation is data collection, an often difficult task involving several trade-offs. We have noted in previous chapters that hardware performance counters have distinct potential to make accurate and precise data collection relatively easy, though they currently do not support many features. In this section, we motivate and propose a series of improvements for on-chip data collection.

7.2.1 Introduction

Collecting data about programs' execution tends to be difficult, often requiring the use of simulators or emulators to get detailed information. The one exception to this rule is sampling performance counters, which make finding program hot spots relatively easy. While the basic idea of performance counters – hardware support for detailed system monitoring – has much promise, existing performance counters can be difficult to use for anything but their intended purpose. The reason for this state of affairs is that processor companies (Intel, most importantly) view performance optimization to be the only important application of program analysis.

As some of the projects in this dissertation have shown, however, there are a variety of potential uses for various pieces of information about program behavior. As the LiMiT project's case studies showed, precise measurements using performance counters alone can reveal interesting, useful data. In fact, all of the projects in this dissertation used program behavior information, the collection of which was more difficult than should have been necessary:

Side-Channel Vulnerability Factor In SVF (Chapter 4), we collected two traces with which to compute leakage through a system. Our study involved changing many microarchitectural parameters, so we required the use of a simulator. However, we would also like to be able to measure SVF on a real system. Collecting the attacker trace would be relatively easy as all we would need to do is execute an attacker. However, recording the victim trace may be difficult. In our study, we used victim memory accesses, the collection of which would create far too much overhead in software. Were there hardware support for monitoring the memory access trace, we could measure the same SVF on a real system.

Approximate Graph Matching In the Centrifuge project (Chapter 5), we used some dynamic information about functions in their clustering. Most prominently, we used dynamic data flow graphs as one of our models. Collecting the data flow graphs required the use of PIN, and had very high overheads when running the programs. As a result, we used small inputs to ensure the benchmarks did not execute for very long.

Malware Detection The malware detection project (Chapter 6) used existing performance counters to collect data. However, we had to use software interrupts to do the actual data collection, limiting the granularity at which we could monitor software without introducing large overheads and thus distorting the data. Ideally, we would have been able to configure the performance counters to automatically send periodic samples to a separate hardware unit or stream into memory.

It's not just research projects which could benefit from better hardware support for program monitoring. Many often-used applications rely on these data:

Valgrind is a tool which tracks memory allocations, loads, and stores in order to find bugs. While its most frequent use is identifing memory leaks, it also catches accesses to invalid memory locations, loads from uninitialized locations, and several other common memory-related programming mistakes. It is generally considered good practice to run it on all C/C++ programs, however its overhead often makes it difficult to use. The overhead is a result dynamic monitoring of the application being run – since there is no hardware monitoring support, Valgrind must either emulate the processor or use dynamic binary translation to write instrumentation into the program.

PIN is a library from Intel which allows programmers to instrument programs with arbitrary monitoring code [107]. It can be used to record full program traces, count dynamic instructions, monitor memory accesses, and many other things. It is very commonly used in program analysis research, however it can often have a very high runtime overhead, often making it inappropriate for production use.

Security often relies on monitoring programs' behavior. Several security proposals involve dynamic monitoring of fine-grained program behaviors, which can create too much overhead to be considered acceptable. Dynamic auditing could catch deviations from normal operation, indicating the process has been exploited. Schemes which monitor control and data flow to identify malware could be more effective than existing antivirus, which primarily use static analysis.

Given the usefulness of program behavior data, in this chapter we propose a set of hardware modifications to expose a variety of information about programs' behavior. In the first section (7.2.2), we give several very simple modifications and discuss some of AMD's recommendations for enhanced performance counting. In the second section (7.2.3), we outline a more comprehensive framework for data collection which we argue would be useful, but would require extensive changes to existing processor designs.

7.2.2 Better Precise Performance Counting

Some modest modifications to existing performance monitoring hardware can reduce the complexity and overheads of precise counting with tools like **LiMiT**. The operations suggested below will reduce **LiMiT**'s read routine from five instructions down to one and reduce the overhead of frequent counter usage patterns. Such low overheads would encourage programs to self-monitor and adapt to changing conditions.

Enhancement #1: 64-bit Reads and Writes **LiMiT**'s overflow handling is necessitated by a lack of full 64-bit read and write support. With 31-bit counters, the counters can overflow every 0.72 seconds, but with 64-bit support they would require centuries to overflow. More recent Intel processors allow full 48-bit writes to performance counters, reducing the number of overflows, but not removing the problem entirely. Until such simple support can be added **LiMiT** will have a vital role in low overhead precise performance measurement.

Enhancement #2: Destructive Reads When characterizing code segments, a difference in counts between two points in the program is often required. A destructive read instruction – one that zeros the counter after reading it – could eliminate the currently necessary subtraction in many cases when counters are used.

Enhancement #3: Combined Reads Currently, the x86 performance counter read instruction requires that the `%ecx` register contain the number of the counter to be read. Were this integrated into the instruction as an immediate, another instruction would be eliminated.

A further proposal for hardware support is AMD's Lightweight Profiling [5]. LWP adds a huge amount of flexibility to performance counters, and would significantly reduce the overhead of self-monitoring were it to be implemented. Among the most interesting features of LWP:

- LWP allows the user process access to all necessary control structures so zero interactions with the operating system are necessary, aside from proper save and restore support for these control registers (which LWP also facilitates).
- LWP adds an new instruction which tells the performance counters to record their contents along with some other configurable information (like the current instruction address) and put these data in a ring buffer. This further reduces the inaccuracy of precise measurements as it delays any necessary data processing and storage overhead away from the region being measured.

7.2.3 A Comprehensive Data Collection Framework

To support a variety of applications, we propose adding an "Enhanced Monitor Unit" (EMU) to each core in a system. The EMU would be capable of monitoring instruction retirement, memory accesses, and the existing performance counters. This would allow it to access any potential piece of architectural data in addition to existing microarchitectural performance event counts.

After filtering events to include only those of interest to the user, the EMU would emit a stream of events to one of several possible destinations, as shown in Figure 7.2. We propose that the EMU be capable of targeting both on- and off-chip destinations to facilitate a variety of usages. For instance, for applications requiring only a small amount of processing, one may want to use another core on the same chip to do data analysis, an option which may be ideal for the malware detection discussed in Chapter 6. Alternatively,

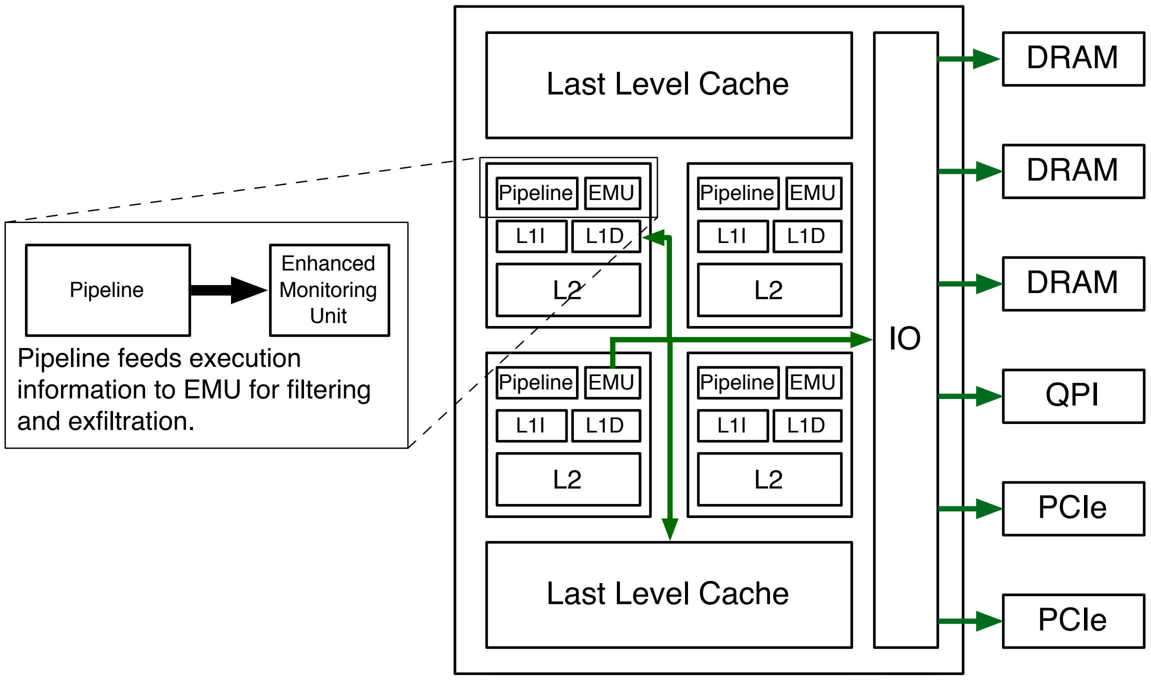


Figure 7.2: An enhanced monitoring unit (EMU) would be capable of monitoring a variety of events. Users could then configure it to filter the events to only those of interest and send the events stream to a destination. Possible destinations include memory for later analysis, other cores for online analysis by software, or off-chip via PCIe to be handled by custom devices.

some EMU use cases may be high-bandwidth and require high-bandwidth storage or on the fly processing. For those tasks, data could be streamed to a custom device (like an FPGA) via PCIe as proposed by Tiwari *et al.* [148].

We propose that the EMU emit data on a separate data bus and be capable of sending it to either other cores on chip or off-die. The separate bus is necessary so that the operation of the EMU does not impact the normal operation of the application which is being monitored. As an alternative, if the EMU's traffic requirements are low enough, it might be able to piggy-back on the cache hierarchy interconnect (but *not* the caches themselves); however that interconnect would have to ensure that the EMU traffic is given a lower priority.

Since the bandwidth which the EMU may require appears to be key in its design, we examine potential event stream bandwidths in Table 7.1. These data indicate that some event streams are simply too high-bandwidth for any destination to keep up with. However, in many cases the maximum possible bandwidth is much higher than a more reasonable expectation. For instance, it may be useful to monitor the full stream of retired instruction addresses. Since modern processors can retire up to about four instructions per cycle at 2 GHz and instruction addresses are 8 bytes, this stream could require a 64 GB/s link – too high for nearly any data sink or interconnect to handle. However, the EMU actually only needs to transmit an instruction address whenever a branch is taken. Also, processes almost never reach their peak speed; typically they achieve a bit more than 1 IPC. Taking those factors into account, the instruction address stream is more likely to require only 3.2 GB/s – a managable bandwidth, even over a PCIe link. The same is true of several other event streams. Accordingly, it is likely feasible to pipe performance data off-chip for analysis, reducing or eliminating the need for on-chip software to do it, thereby reducing perturbation.

We have not evaluated the area impact of providing the EMU or EMU interconnect on a processor, so it is unclear how much cost it would add. However, if this cost is determined to be low, we believe it would be beneficial to add EMUs to future processors.

Event	Events / Cycle	Max B/W at 2GHz
Retired Instruction Addresses	4 maximum	64 GB/s
with average basic block size of 5 and IPC of 1	0.2 expected	3.2 GB/s
Memory Load/Store Addresses	2 max	32 GB/s
with load/store density of $\frac{1}{4}$ and IPC of 1	0.25 expected	4 GB/s
Performance Counters	4 maximum	64 GB/s
sampling at 100 cycle period	0.04	0.64 GB/s
All 64-bit instruction results	4 maximum	64 GB/s
with IPC of 1	1 expected	16 GB/s

I/O Method	Bandwidth
PCIe Gen3 x16	15.75 GB/s
PCIe Gen4 x16	31.51 GB/s
DDR3 Single Channel	24 GB/s
Intel QPI	32 GB/s

Table 7.1: Event streams which the EMU should handle and necessary bandwidths for each assuming a microarchitecture similar to an Intel Nehalem.

Chapter 8

Concluding Remarks

In this dissertation, we have presented four case studies in the use of advanced measurement and analysis. We have also argued for more quantitative analysis techniques, better hardware support for measurement, and software tools for data management and analysis. We conclude with a summary of the dissertation and two notes: first, some lessons learned. Second, an argument *against* our main thesis of quantitative methods.

8.1 Summary & Contributions

LiMiT In Chapter 3 we introduced a new method of using existing performance counters to measure code regions precisely. Our novel method requires only 11ns to read counters, 23x faster than existing interfaces – reducing perturbation, thus increasing accuracy – while still preserving thread isolation. As a result of our low overhead technique, we were able to conduct several case studies examining the behavior of fine-grained locks and system library calls in production software like Firefox, Apache, and MySQL. We found that production applications differ significantly from the existing, popular Parsec [18] benchmark suite, motivating a new suite or methods to avoid using benchmarks suites entirely. We also demonstrated the value and potential of hardware-based measurement systems for collecting data both precisely and accurately, two attributions which otherwise must often be traded-off with each other.

SVF In Chapter 4 we introduced a framework for defining metrics for side-channel security. Side-channel information leaks have been demonstrated to reveal information about secrets like encryption keys and encrypted data, making them potentially very dangerous. While both attacks and defenses have been published, there is no clear way to measure the efficacy of defenses or systems' vulnerability to attack. SVF takes a black box, experimental approach and defines this vulnerability as the leakage of computationally recognizable patterns through the system. In defining SVF in a very generic manner with few assumptions and little intuition about the system being measured, we found many surprises about cache side-channel leakage via a case study. In fact, it seems that "cache side-channel" is often a misnomer. We conclude that side-channel information leakage a system-level properly and very difficult to intuitively reason about, motivating quantitative methods further.

CENTRIFUGE In Chapter 5 we introduced a technique for approximately clustering program graphs. This technique can be used to assist in program comprehension by finding and grouping functions which are similar in terms of their program graphs. In a case study of functions in SPEC 2006 [67], we found that functions clustered based on their dynamic data flow graphs also react similarly (in terms of performance) to advanced compiler optimizations. Perhaps more interestingly, we found examples of functions which are very similar – based on their program graphs and optimization reactions – but whose code does not appear at all similar. These examples represent similarities which would not have been caught by manual inspection.

Malware Detection In Chapter 6 we introduced methods to use standard machine learning algorithms on performance counter data to learn the behaviors of malware and detect new versions of them. This system has two advantages: First, because it is dynamic, behavior based detection, it is able to detect new variants of old malware. Second, the methods we used were entirely automated. This is in contrast to production antivirus systems, which are typically manually written softwares composed of millions of lines of code, the bugs in which can sometimes be used to subvert the AV protection. Although the results of our case study were not sufficiently positive to replace existing AV systems, we conclude that automated learning techniques have promise.

8.2 Lessons Learned

Support for Measurement The **LiMiT** project discussed in Chapter 3 demonstrated the utility of performance counting beyond traditional hot-code profiling. Its technique for reading the counters significantly reduced the overhead of using existing counters for precise instrumentation of code. The malware detection project in Chapter 6 also used performance counters differently than their intended purpose. In both cases, the overhead of using counters was relatively low (compared to other techniques), so inaccuracy was also kept relatively low. Indeed, hardware support for measuring both software and hardware is likely the only way to shift the accuracy-precision trade-off curve we discussed in Chapter 2. To that end, we would like to see better support; in Section 7.2.3 we outline a system which would allow measuring a larger variety of characteristics at an even finer granularity with less perturbation and thus lower inaccuracy. Such a system may be useful beyond research, specifically in security for software behavior auditing.

Unbiased, Black Box Investigation When we started building the simulator used in SVF (Chapter 4), we at first set out to build a cache-only simulator – why bother with more when we are investigating *cache* side-channel attacks? However, we ended up concluding that side-channel leakage has to be investigated in a system context; hardware systems have so many moving parts that it is never clear exactly which is causing a particular phenomenon, if it can even be attributed to a single part. In the simulated system we built, the pipeline turned out to leak just as much as the cache. It wasn't far from pure serendipity which caused us to even model the pipeline in the simulator. And therein lies the lesson. Without an unbiased, quantitative investigation like SVF allowed, this phenomenon would likely not have been considered. By treating the hardware like a black box and poking it like a young child scientist pokes an ant, a new possibility in our thinking arose. Very little investigation of man-made systems (like computer hardware) use this approach, but it has value.

Unbiased, Automated Investigation Two of the projects presented in this dissertation relied on automated methods for analysis. In the CENTRIFUGE project of Chapter 5, we presented a technique to cluster similar functions based on approximate graph matching. The results were quantitatively good, but many of the resulting clusters were qualitatively unintuitive. Similarly, the malware detectors shown in Chapter 6 used labeled data, but no addition intuition was used in their training. We somewhat surprisingly found that families of applications could be identified (with reasonable accuracy) based entirely on a large collection of low-dimensional vectors – no approximate instruction matching, static analysis, or eyeballing required. In both cases, the unintuitive or surprising results occurred only because we used automated methods. While automation enabled both studies in dealing with the sheer scale of data, it also served another, more important purpose: Automation allowed us to remove as much personal bias as possible.

Informing Qualitative Understanding It is not always possible to completely solve a problem using quantitative methods. Some problems are not easily quantified, some do not yet have methods which can be applied. Conversely, not all quantitative methods necessarily have a well defined problem or need be constrained to the problem for which they were developed. SVF is not yet a complete solution for measuring leakage – it requires some knowledge about the method of attack. We have used **LiMiT** many times to take interesting measurements which did not have practical use. Phase analysis [134] – used by both SVF and our malware detection project – reveals very interesting program behavior; yet, its stated use would constrain it to finding representative samples of within a program's execution. CENTRIFUGE merely organizes code and does not directly result in useful new innovations, yet we gained several interesting new insights.

We argue for a new problem for which all quantitative methods should be considered partial solutions: comprehension. What are programs doing? How does hardware work? Understanding some new aspect, gaining some interesting insight, or otherwise informing one's intuition are worthy goals in and of themselves, even if this new knowledge has no immediate application or impact.

8.3 An Argument Against Quantitative Methods

This dissertation has argued strongly for the use of quantitative methods whenever possible, eschewing intuition as much as possible. Quantitative methods are a core tenet of science; they have led the way from alchemy and bloodletting to material sciences and modern medicine. Our thesis, therefore, is an argument with which few would likely disagree. So allow us to disagree and make an argument against quantitative methods:

Relying entirely on quantitatively driven investigation can lead to incrementalism. Take, for example, the Copernican Revolution. When he introduced it, Copernicus' heliocentric model of the cosmos was not able to produce more accurate predictions of planetary positions than the Ptolemaic model it was intended to replace. In fact, as observations of planetary positions grew in accuracy, the Ptolemaic model was able to correct for inaccuracies by making small, incremental changes – by adding more epicycles, for instance [96]. While fundamentally more accurate, Copernicus' model could not be quantitatively justified during its development; were he entirely data driven, Copernicus would not have published his model.

In computer architecture, with the ability to use advanced measurement and analysis tools to identify faults and bottlenecks in existing systems, it will become easier to find opportunities for improvements. Many of these opportunities will likely be fixed with relatively small modifications: a better heuristic for a predictor, a new instruction for a new application. These improvements are fine and likely to yield improvements. They are easy to quantitatively motivate and it is thus equally easy to justify their research and development costs. However, small changes – by their nature – can do little more than play at the margins. And so quantitative methods can become a trap.

One might think of computer architecture as a high-dimensional optimization problem. Exploring this space based *entirely* on rigorous quantitative methods is akin to optimizing locally; similar to using gradient descent. The trap is being caught in a local minimum. The only way out is to do something radically different, to create innovations which can rarely be entirely quantitatively and soundly justified before investigating them. These extreme new creations can explore totally different areas of the architecture design space, thus escaping known minima; however, the probability of quickly finding better architectures is low and the cost of experimenting in new spaces is high. These are risky propositions. The choice of where to explore must be guided in some manner – random search in high-dimensional space is unlikely to yield good results.

In radical innovation, therefore, there must be some element of creativity. There must be some subjective, qualitative, experienced, intuitive guide. This does not mean, however, that quantitative methods have no role to play. The ability to quickly measure, analyze, model, reason, and quantitatively understand software and architectures is the key to developing better intuition and rapidly evaluating new ideas. New and better quantitative methods, therefore, need not inhibit or discourage radical innovation and need not lead to incrementalism. Rather, new quantitative methods can teach us and better inform our intuition. Some may even serve very little purpose besides assisting in uncovering new insights. Quantitative methods can help scale the intuition wall.

Part IV

Glossary & Bibliography

Glossary

accelerator A dedicated hardware unit which is designed to run one or a small class of algorithms at higher speed and/or higher energy efficiency than general purpose processors.

antivirus A system which detects and protects users from malware.

artificial neural network A non-linear supervised machine learning method which makes predictions by propagating inputs through a graph of "neurons" which transform and combine values based on connection weights and activation functions.

basic block A linear sequence of instructions with a single entry point and a single exit point.

BJT Bipolar Junction Transistor. Both a type of transistor and an early silicon manufacturing & design technology which used BJTs to implement logic gates.

branch predictor A hardware structure which predicts which direction an branch instruction (*i.e.,* an 'if' in code) will take. Used to avoid stalling processor pipelines.

branch target prediction A hardware structure which determines the address to which a processor must jump if a branch is taken.

cache A processor cache is a small memory which stores recently accessed data. Used to speed up memory accesses in the common case.

cache line A block of storage within a cache.

cache set A set of cache lines which are all associative with each other. *i.e.,* the memory addresses associated with all of the lines all hash to the same index.

classifier A type of machine learning methods which predict the class (*i.e.,* category) of data.

clustering *verb.* The act of grouping similar items. *noun.* A group of similar items placed together by a clustering algorithm.

CMOS Complimentary metal-oxide-semiconductor. A silicon manufacturing & design technology which uses both NMOS and PMOS transistors to implement logic gates. By using complimentary transistor sets for each gate, it obtains both high stability/reliability and low power dissipation.

control/data flow graph A type of program graph which represents both control and data dependencies.

dark silicon Term given to describe the idea that only a fraction of a modern silicon chip can operate simultaneously as a result of fixed power budgets. See "Dark silicon and the end of multicore scaling" [50] .

decision tree A supervised machine learning method which clusters training data into a binary tree. Issues predictions from input vectors by traversing the tree to a leaf and combining the training data found in the leaf.

dynamic data flow graph A type of program graph which represents only data dependencies observed during one or more executions of the program.

feature A measurement or property of data which can be used as input or part of an input to a machine learning algorithm. Typically numeric.

feature vector A fixed-length vector of features which is used as input to machine learning algorithms.

front-end The front-end of a processor pipeline is the set of hardware units concerned with supplying instructions for execution. It includes the instruction cache and instruction decoder.

hashing function A function which determines in which cache set the data for a particular memory address must be stored.

instruction mix The breakdown of types of instructions within a set of instructions. *i.e.,* a vector which contains the fractions of memory instructions, floating point instructions, integer arithmetic instructions, et cetera.

intermediate representation Also known as "intermediate language". A abstract, general language used internally to compilers to describe computer programs. Typically include fewer constructs than source code languages. Typically used in compiler analysis and transformation passes. Example: the LLVM IR [100].

interval Subsequence of an oracle or side-channel trace. The symbols within the subsequence are combined to form a new symbol, creating a shorter trace of intervals, possibly in a new alphabet.

KNN k-Nearest Neighbors. A supervised machine learning method which predicts based on the geometric proximity of input data to data in a training set.

malware Malicious software. Includes viruses, adware, worms, et cetera. See Table 6.1.

memory disambiguation The act of determining whether or not two memory operations access the same memory address. Used to allow multiple in-flight memory operations.

microarchitectural Pertaining to the microarchitecture of an architecture or instruction set architecture (ISA) rather than its logical (architectural) state.

microarchitecture The implementation of a particular computer architecture (or ISA). The microarchitecture is typically hidden from the programmer and thus can only affect software performance rather than correctness, assuming the microarchitecture contains no bugs. For example, caches are a microarchictural component – they serve to speed up applications, but their state cannot be directly accessed by programs.

NMOS N-Channel MOSFET. Both a type of transistor and a silicon manufacturing & design technology which used NMOS transistors to implement logic gates.

performance counter Register built into modern microprocessors which can be configured to count the occurrences of particular events (*e.g.,* cache misses). Synonymous with “hardware performance counter”.

phase Portion of program’s execution with a set of consistent characteristics (*i.e.,* cache misses, or branching behavior). Phases tend to repeat themselves during the execution of programs and have the same characteristics when they do so. Phases have been shown to be strongly correlated to the set of program code running during their execution [134].

PMOS P-Channel MOSFET. Both a type of transistor and a silicon manufacturing & design technology which used NMOS transistors to implement logic gates.

prefetcher A hardware unit which predicts memory addresses which will be accessed by software in the near future then pre-loads those data locations into caches.

program analysis See program comprehension.

program comprehension The act of understanding or analyzing software programs to understand their general operation or a specific property.

program graph A graph which represents some aspect of program code. Includes graphs like dynamic data flow graphs and control/data flow graphs.

random forest An ensemble machine learning method which uses a set of decision trees all built with some element of randomness.

ROC curve Receiver Operating Characteristic curve. A plot which shows the trade-off between false positive and true positive rates.

rootkit A piece of malicious software which hides activities or processes.

set associativity The number of cache lines within each cache set.

side-channel Channel through which side-channel information leakage can occur.

side-channel information leakage Data which is otherwise private can be "leaked" to non-privileged due to unintended side effects of programs operating on the data. See Chapter 4.

SMT Simultaneous Multi-Threading. Hyper-Threading is Intel's proprietary version. A CPU technology which exposes a single CPU core as multiple logical cores and allows multiple threads to execute on the same pipeline simultaneously. Used to increase utilization of pipeline resources.

TLB Translation Lookaside Buffer. A hardware structure which caches virtual address mappings. Used to speed up virtual to physical address translation in the common case.

trace A finite sequence of symbols representing logical events in either a victim's execution or observations an attacker has made.

Acronyms

APK application package file.

AVF Architectural Vulnerability Factor.

CDF cumulative distribution function.

CPI cycles per instruction.

CSV Cache Side-channel Vulnerability.

FPGA field programmable gate array.

GPU graphics processing unit.

HTTP hypertext transfer protocol.

HTTPS HTTP secure.

IFT information flow tracking.

IPC instructions per cycle.

ISA instruction set architecture.

LLVM Low Level Virtual Machine.

ML machine learning.

MTBF mean time between failures.

MTTF mean time to failure.

NRE non-recurring engineering.

OLTP online transaction processing.

PCC Pearson Correlation Coefficient.

PCIe Peripheral Component Interconnect Express.

RMS recognition mining synthesis.

SSA static single assignment.

SVF Side-channel Vulnerability Factor.

UML unified modeling language.

URL uniform resource locator.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, 1999.

[3] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, C-W Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 2–9. IEEE, 2005.

[4] Lars Alvincz and Sabine Glesner. Breaking the curse of static analyses: Making compilers intelligent via machine learning. In *SMART'09 Workshop*, 2009.

[5] AMD. AMD64 technology lightweight profiling specification, revision 3.08, 2010.

[6] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[7] Manuel Arenaz, Juan Touriño, and Ramon Doallo. XARK: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst.*, 30:32:1–32:56, October 2008.

[8] *ARM Architecture Reference Manual, Arm v7-A and ARMv7-R edition, Errata markup.* ARM DDI 0406B_errata_2011_Q3 (ID120611).

[9] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *Parallel Computing in Science and Engineering*, pages 61–88, 1987.

[10] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proc. of the 18th ACM conf. on Computer and communications security*, pages 375–388, New York, NY, USA, 2011. ACM.

[11] Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 141–153. IEEE, 2009.

[12] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proc. of the 10th intl. conf. on Recent advances in intrusion detection*, RAID'07, pages 178–197. Springer-Verlag, 2007.

[13] Hamid Abdul Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering*, 35(4):497–514, July/August 2009.

[14] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, 2009.

[16] Hugues Berry, Daniel Gracia Pérez, and Olivier Temam. Chaos in computer performance. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 16(1):013110–013110, 2006.

[17] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. *SIGPLAN Not.*, 27:223–233, September 1992.

[18] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[19] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proc. of the 2012 ACM conf. on Computer and communications security*, pages 833–844, 2012.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[21] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 318–329. IEEE Computer Society, 2008.

[22] BlackHat Library. Jynx rootkit2.0, Mar 2012.

[23] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265 – 275, 23-26 2003.

[24] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring Pay-per-Install: The commoditization of malware distribution. In *Proc. of the 20th USENIX Security Symp.*, 2011.

[25] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[26] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34, New York, NY, USA, 2006. ACM.

[27] J. Cavazos. Intelligent compilers. In *Cluster Computing, 2008 IEEE International Conference on*, pages 360 –368, sept. 2008.

[28] David A Chapin and Steven Akridge. How can security be measured. *information systems control journal*, 2:43–47, 2005.

[29] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan 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.

[30] Licheng Chen, Zehan Cui, Yungang Bao, Mingyu Chen, Yongbing Huang, and Guangming Tan. A lightweight hybrid hardware/software approach for object-relative memory profiling. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 46–57. IEEE, 2012.

[31] Eric Chien, Liam OMurchu, and Nicolas Falliere. W32.Duqu: The Precursor to the Next Stuxnet. In *Proc. of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2012.

[32] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proc. of the the 6th joint meeting of the European software engineering conf. and the ACM SIGSOFT symp. on The foundations of software engineering*, ESEC-FSE '07, pages 5–14, 2007.

[33] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[34] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Nathan Clark, Amir Hormati, Scott Mahlke, and Sami Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 147–157, New York, NY, USA, 2006. ACM.

[36] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: A science-oriented DBMS. *Proc. VLDB Endow.*, 2(2):1534–1537, August 2009.

[37] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[38] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 353–364, New York, NY, USA, 2011. ACM.

[39] John Demme and Simha Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, January 2012.

[40] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *The 39th Intl. Symp. on Computer Architecture*, pages 106–117, 2012.

[41] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th annual international symposium on Computer architecture*, ISCA '13, New York, NY, USA, 2013. ACM.

[42] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 245–257, New York, NY, USA, 2003. ACM.

[43] Leonid Domnitser, Nael Abu-Ghazaleh, and Dmitry 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.

[44] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, New York, NY, USA, 2007. ACM.

[45] Tudor Dumitras and Darren Shou. Toward a standard benchmark for computer security research: the worldwide intelligence network environment (WINE). In *Proc. of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 89–96. ACM, 2011.

[46] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 293–302. IEEE, 2008.

[47] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[48] Lieven Eeckhout. Computer architecture performance evaluation methods. In *Synthesis Lectures on Computer Architecture*. Morgan Claypool, 2010.

[49] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the vax-11/780. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 301–310, New York, NY, USA, 1984. ACM.

[50] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[51] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M Blackburn, and Kathryn S McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 319–332. ACM, 2011.

[52] Marius Evers, Sanjay J Patel, Robert S Chappell, and Yale N Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 52–61. IEEE Computer Society, 1998.

[53] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. *SIGARCH Comput. Archit. News*, 38:362–370, June 2010.

[54] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):3, 2009.

[55] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[56] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[57] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *Proc. of the 1996 IEEE Symp. on Security and Privacy*, pages 120–135, 1996.

[58] Linux Foundation. Linux kernel 2.6.32, `perf_event.h`.

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

[60] Grigori Fursin and Olivier Temam. Collective optimization. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[62] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 321–330, New York, NY, USA, 2008. ACM.

[63] David Gullasch, Endre Bangerter, and Stephan 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.

[64] M. Gupta, R. Singh Rao, and A.K. Tripathi. Design pattern detection using inexact graph matching. In *Communication and Computational Intelligence (INCOCCI), 2010 International Conference on*, pages 211 –217, dec. 2010.

[65] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, volume 10, pages 37–47, 2010.

[66] Kim Hazelwood. Dynamic binary modification: Tools, techniques, and applications. *Synthesis Lectures on Computer Architecture*, 6(2):1–81, 2011.

[67] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[68] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269, New York, NY, USA, 2010. ACM.

[69] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[70] Michael J. Hind, Vadakkedathu T. Rajan, and Peter F. Sweeney. Phase shift detection: A problem classification, 2003.

[71] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[72] Amir Hormati, Nathan Clark, and Scott Mahlke. Exploiting narrow accelerators with data-centric subgraph mapping. In *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, pages 341 –353, 11-14 2007.

[73] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 114–122, New York, NY, USA, 2006. ACM.

[74] Joel Hruska. Nvidia deeply unhappy with TSMC, claims 20nm essentially worthless.

[75] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 209–220. IEEE, 2002.

[76] *Intel 64 and IA-32 Architecture Software Developers Manual*, volume 3B: System Programming Guide, Part 2.

[77] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 39–50. IEEE, 2008.

[78] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 359–370, 2006.

[79] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last level cache (llc) performance of data mining workloads on a cmp-a case study of parallel bioinformatics workloads. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 88–98. IEEE, 2006.

[80] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security and Privacy*, pages 80–94, 2012.

[81] Wayne Jansen. *Directions in security metrics research.* DIANE Publishing, 2010.

[82] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 197–206. IEEE, 2001.

[83] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L.K. John. Measuring benchmark similarity using inherent program characteristics. *Computers, IEEE Transactions on*, 55(6):769 –782, june 2006.

[84] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[85] Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *Advances in Cryptology–ASIACRYPT 2009*, pages 703–720. Springer, 2009.

[86] Stefanos Kaxiras and Margaret Martonosi. Computer architecture techniques for power-efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[87] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. *SIGARCH Comput. Archit. News*, 26(3):15–26, 1998.

[88] R.A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *Selected Areas in Communications, IEEE Journal on*, 7(4):448–457, 1989.

[89] Cetin Kaya Koc. *Cryptographic Engineering.* Springer Publishing Company, Incorporated, 1st edition, 2008.

[90] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis.* Springer-Verlag, 1999.

[91] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang 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.

[92] Kostas Kontogiannis. Program representation and behavioural matching for localizing similar code fragments. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 194–205. IBM Press, 1993.

[93] J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309, Stuttgart, October 2001.

[94] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, 2007.

[95] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, pages 83–97, 1955.

[96] Thomas S Kuhn. *The structure of scientific revolutions.* University of Chicago Press, 1962.

[97] Laboratory of Cryptography and System Security (CrySyS Lab). sKyWIper: A Complex Malware for Targeted Attacks. Technical Report v1.05, Budapest University of Technology and Economics, May 2012.

[98] R. Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.

[99] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proc. of the 17th ACM conf. on Computer and communications security*, pages 399–412, 2010.

[100] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See http://www.llvm.org.*

[101] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183, 2010.

[102] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.

[103] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. A data mining framework for building intrusion detection models. In *In IEEE Symposium on Security and Privacy*, pages 120–132, 1999.

[104] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and improving operating system effects in control flow prediction. *SIGPLAN Not.*, 37:68–80, October 2002.

[105] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[106] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 500–511. IEEE Press, 2012.

[107] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[108] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proc. of the sixth ACM workshop on Scalable trusted computing*, pages 71–76, 2011.

[109] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 118–129. IEEE Press, 2012.

[110] Fontanini Matias. Linux rootkit implementation, Dec 2011.

[111] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, 2003.

[112] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In *18th International Conference on Data Engineering*, pages 117 – 128, 2002.

[113] Thomas S. Messerges, Ezzy A. Dabbish, and Robert 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.

[114] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 66–72, New York, NY, USA, 2001. ACM.

[115] Shirley Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In Peter Sloot, Alfons Hoekstra, C. Tan, and Jack Dongarra, editors, *Computational Science – ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 904–912. Springer Berlin / Heidelberg, 2002.

[116] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 29–40. IEEE, 2003.

[117] Todd Mytkowicz, Amer Diwan, and Elizabeth Bradley. Computer systems are dynamical systems. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 19(3):033124–033124, 2009.

[118] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2010)*, October 2010.

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

[120] Oprofile. http://oprofile.sourceforge.net/.

[121] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

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

[123] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[124] Zhelong Pan and Rudolf Eigenmann. PEAK – a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst.*, 30:17:1–17:43, May 2008.

[125] Pearson product-moment correlation coefficient. http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

[126] Colin Percival. Cache missing for fun and profit, 2005.

[127] Perfmon2. http://perfmon2.sourceforge.net/.

[128] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[129] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 412–423, New York, NY, USA, 2007. ACM.

[130] Rabbit, a performance counters library for Intel/AMD processors and linux. http://www.scl.ameslab.gov/Projects/Rabbit/.

[131] Zulfikar Ramzan, Vijay Seshadri, and Carey Nachenberg. Reputation-based security: An analysis of real world effectiveness, Sep 2009.

[132] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proc. of the 5th intl. conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer-Verlag, 2008.

[133] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5 – 19, jan 2003.

[134] 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.

[135] Dr. Richard Sites. Personal communications.

[136] Kevin Skadron, Margaret Martonosi, David I August, Mark D Hill, David J Lilja, and Vijay S Pai. Challenges in computer architecture evaluation. *Computer*, 36(8):30–36, 2003.

[137] Randy Smith and Susan Horwitz. Detecting and measuring similarity in code clones. In *International Workshop on Software Clones (IWSC)*, 2009.

[138] Daniel J Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009.

[139] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[140] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[141] Sal Stolfo, S.M. Bellovin, and D. Evans. Measuring security. *Security Privacy, IEEE*, 9(3):60–65, 2011.

[142] Brett Stone-Gross, Ryan Abman, RichardA. Kemmerer, Christopher Kruegel, DouglasG. Steigerwald, and Giovanni Vigna. The underground economy of fake antivirus software. In Bruce Schneier, editor, *Economics of Information Security and Privacy III*, pages 55–78. Springer New York, 2013.

[143] Michael Stonebraker, Jacek Becla, David J DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B Zdonik. Requirements for science data bases and scidb. In *CIDR*, volume 7, pages 173–184, 2009.

[144] Student. The probable error of a mean. *Biometrika*, pages 1–25, March 1908.

[145] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

[146] Pter Szr and Peter Ferrie. Hunting for metamorphic. In *In Virus Bulletin Conference*, pages 123–144, 2001.

[147] Songsri Tangsripairoj and M. H. Samadzadeh. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1539–1545, New York, NY, USA, 2005. ACM.

[148] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the potential of program analysis peripherals. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 53–63. IEEE, 2009.

[149] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM, 2009.

[150] Trend Micro Corporation. Russian underground.

[151] Spyridon Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. A framework for unrestricted whole-program optimization. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 61–71, New York, NY, USA, 2006. ACM.

[152] Secil Ugurel, Robert Krovetz, and C. Lee Giles. What's the code? automatic classification of source code archives. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638, New York, NY, USA, 2002. ACM.

[153] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 163–174. ACM, 2011.

[154] Intel vtune. `http://software.intel.com/en-us/intel-vtune/`.

[155] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

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

[157] Zhenghong Wang and R.B. 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.

[158] Joe H. Jr Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, pages 236–244, March 1963.

[159] Dong Hyuk Woo and H-HS Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[160] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proc. of the 2012 42nd Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 1–12, 2012.

[161] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 16–16, 1996.

[162] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B Lee. Side channel vulnerability metrics: the promise and the pitfalls. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 2. ACM, 2013.

[163] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symp. on*, pages 95 –109, may 2012.