3-7-2022

MADFAM: MicroArchitectural Data Framework and Methodology

Tor J. Langehaug [*]
Air Force Institute of Technology

Scott R. Graham
Air Force Institute of Technology

Christine M. Schubert Kabban
Air Force Institute of Technology

Brett J. Borghetti
Air Force Institute of Technology

Recommended Citation

T. J. Langehaug, S. R. Graham, C. M. Schubert Kabban and B. J. Borghetti, "MADFAM: MicroArchitectural Data Framework and Methodology," in IEEE Access, vol. 10, pp. 23511-23531, 2022, doi: 10.1109/ACCESS.2022.3153313.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 3, 2022, accepted February 13, 2022, date of publication February 22, 2022, date of current version March 7, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3153313

MADFAM: MicroArchitectural Data Framework and Methodology

TOR J. LANGEHAUG, (Student Member, IEEE), SCOTT R. GRAHAM, (Senior Member, IEEE), CHRISTINE M. SCHUBERT KABBAN, AND BRETT J. BORGHETTI

Graduate School of Engineering and Management, Air Force Institute of Technology, WPAFB, OH 45433, USA

Corresponding author: Tor J. Langehaug (tor.langehaug@afit.edu)

ABSTRACT In the aftermath of Spectre and Meltdown researchers have proposed a variety of attack detection solutions by applying machine learning to data collected from hardware performance monitoring units. Although many microarchitectural attack detection systems provide high-accuracy detection results, the behavior of the underlying data collection mechanisms is not well described or understood. This research introduces the MicroArchitectural Data Framework And Methodology (MADFAM) to prescribe a systematic approach to collecting and preserving the information available in sequences of microarchitectural data. The proposed framework focuses on hardware performance counters (HPCs) as the primary data source. HPC configuration is complex, which makes it difficult for others to reproduce results or advance the state-of-the-art. This framework includes a description of design decisions that HPC research must consider across an array of problem domains, including information security. MADFAM proposes a data collection architecture and evaluation criteria to improve the discussion about the experimental settings and design decisions used in HPC research. The proposed framework evaluation criteria are then used to establish a baseline characterization of time series data that future research can use to compare alternative framework implementations.

INDEX TERMS Computer organization, computer engineering, hardware security, hardware performance counter, Meltdown, microarchitectural data, Spectre.

I. INTRODUCTION

The release of the Spectre [1] and Meltdown [2] attacks in 2018 awakened the computer security research community to risks posed by unintended microarchitectural side effects resulting from performance optimizations in microprocessors. Since the release of Spectre and Meltdown, over 40 papers have been published [3], [4] exposing different types of speculative or transient execution attacks.

To combat these emerging threats, researchers have proposed secure cache schemes [5], [6], trusted execution environments [7], [8], and Hardware Performance Counter (HPC)-based microarchitectural detectors to identify malicious cache access patterns [9]–[11] or patterns associated with the critical path to exploit a microarchitectural side channel [12]. Secure cache architectures and trusted execution environments provide fundamental long-term solutions to protect systems against microarchitectural threats. However, the architectural changes necessitated by secure caches and trusted execution environments do not address the near-term threat.

An alternative to replacing existing hardware is to leverage microarchitectural data to improve detection mechanisms. The most frequently used source for microarchitectural data are HPCs, which are present on most embedded, consumer, and server-grade microprocessors. These HPCs provide detailed information about the performance of a processor's microarchitecture (e.g., cache hit rate or # of branch mispredictions) during the computer's operation. HPCs are designed to identify performance problems and are able to collect large amounts of data with relatively small impacts to the system performance. Performance Monitoring Units (PMUs), which are discussed in Section II-B, provide HPCs.

Prior research has proposed using these HPCs to detect specific attacks such as Spectre [9] or Meltdown [10], cache attacks [11], and other malware [13] by creating

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

machine learning models to identify known malicious patterns in the microarchitectural data for a specific attack. The effectiveness of this approach is not yet clear. Some research disputes the value [14], [15], though others such as Malone *et al.* [16] and Basu *et al.* [17] present rigorous rationale and statistical reasoning to explain how HPC event sequences create unique execution fingerprints to confirm conformance to the expected Control Flow Graph (CFG) for a known program.

Given several demonstrated successes of HPC-based security detection systems there is reason to continue evaluating the utility of HPCs under different conditions. A common problem associated with HPC research is grappling with the complexities of the configuration settings and data structures required to achieve a desired outcome. This research is motivated by the need to characterize the data collection process in support of applying statistical models and machine learning techniques. Additionally, the framework benefits HPC machine learning system requirements for labeled data, structured input, an interface to make online inferences, and reducing random error. The framework presented in this research provides a structured terminology to discuss settings and an implementation to achieve these beneficial properties. This research makes the following notable contributions:

- Organizes typical data collection considerations to link HPC configuration with machine learning goals.
- Proposes a novel framework that incorporates new design ideas for a data collection architecture that can be expanded or improved upon in future research.
- Provides an analysis approach using three different microprocessor architectures to establish reference data that future HPC research and data collection tools could use for comparison.
- Highlights differences in HPC behavior between processor generations and vendors which presents challenges to general purpose machine learning models.

The remainder of this paper is structured to provide a broad context for HPC research. Section II provides background to relate several research areas, including computer architecture, hardware sensors, performance counter data collection, cache side channels, and transient execution. Section III discusses related research emphasizing HPC data for information security applications. Section IV introduces a model to describe the execution environment from which data is gathered and organizes several parameters researchers must consider. Section V describes an implementation of the framework to collect labeled data that is suitable for supervised machine learning research. Section VI presents four evaluation criteria to describe the behavior of the implemented architecture. Section VII discusses the evaluation criteria results and highlights computer system behavioral challenges that machine learning researchers should consider when using HPC data. Finally, Section VIII summarizes the results and discusses directions for future work.

II. BACKGROUND

This section provides a summary of relevant information related to computer system organization, hardware sensors or data sources, PMUs, HPC configuration, the motivating hypothesis underlying this work, cache side channels, and transient execution attacks. Each of these topics is considered in contemporary research and the treatment given here is intentionally broad. Interested readers are invited to consult the works referenced in Section III for detailed information.

A. TWO PERSPECTIVES ON SYSTEM ARCHITECTURE

An overview of the environment in which programs execute or run provides a foundation to understand where microarchitectural data originates and why the data is of research interest. Figure 1 presents a simplified view of the computing system, which may be read from the bottom-up to understand the computer architect's perspective. Alternately, Figure 1 may be read from the top-down to understand a user's perspective of the computer system.

One challenge faced by computer architects is to organize the computer's memory system to provide users with the perception that the high-speed Central Processing Unit (CPU) has access to a large and fast storage device. Small memory, such as the registers inside the CPU core, are very fast. However, these few instances of small memory are insufficient for the data storage requirements of modern applications and are cost-prohibitive to expand. Larger memory devices, such as Dynamic Random-Access Memory (DRAM) or disk-based storage systems, have data access times several orders of magnitude slower than CPU registers. Hierarchies of caches are a design decision made by computer architects to move data that was used in the recent past or is likely to be used in the near future as close to the CPU as possible to facilitate the user's perception of a large and fast storage system. A serious concern for information security is that several user programs share these resources simultaneously, and this contention can lead to information leakage.

User applications, programs, and services such as databases or web servers rely on an operating system to efficiently manage access to resources while ensuring private or secret information confidentiality. In most cases, users are unaware of the complex multitasking performed by both the operating system and hardware to create the perception of fast in-order execution of programs written in a high-level language. Even programs that use parallel task execution (i.e., multithreading or multiprocessing) to reduce execution time do so with limited knowledge about how the operating system decides which tasks to execute or how the underlying hardware executes the instructions.

B. HARDWARE SENSORS

Contemporary computing systems, especially mobile devices, are equipped with different types of sensors, often added to a computing device for a specific purpose. For

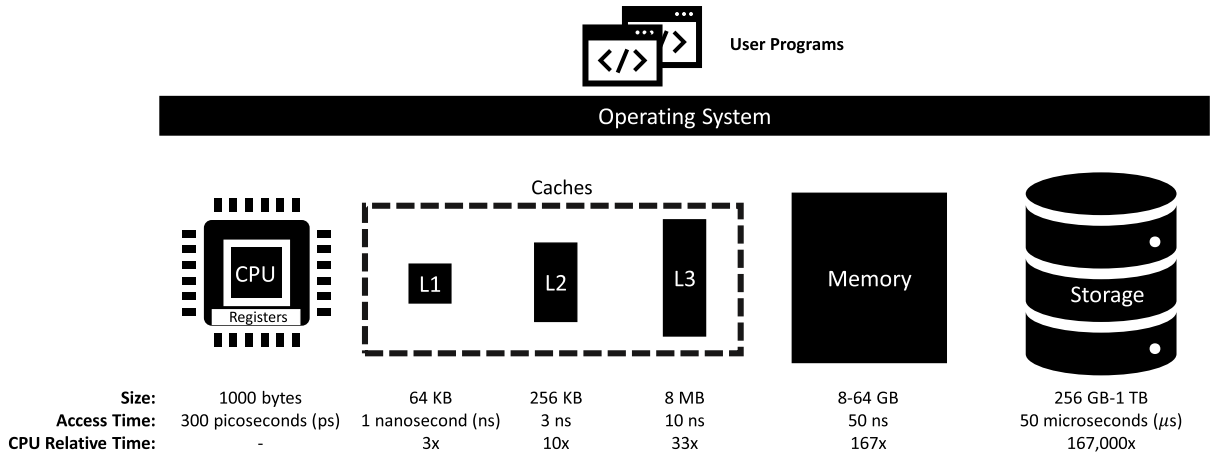


FIGURE 1. Overview of computer organization (timescales based on [18]).

example, processors must balance the power used by the CPU with performance requirements. Thermal and voltage sensors on the CPU die or package provide continuous streams of data to control logic which makes decisions about the CPU's current maximum operating frequency. In the case of thermal sensors, logic on the CPU package may prevent exceeding thermal design limits for the CPU whereas an operating system may use thermal sensors to increase system fan speeds as specified by the user. Spinning disk hard drives in laptops serve as another example of hardware sensors. Mobile hard drives contain internal accelerometers that sense when a device is falling so that drive rotation can be halted to prevent catastrophic data loss on impact.

CPUs are composed of smaller components or modules and the term *microarchitecture* refers to the organization of sub-components such as caches, buffers, and execution units within a CPU implementation. PMUs are a hardware facility that encapsulates microarchitectural sensors. PMUs are common on several device classes including smartphones, laptops, desktops, servers and embedded devices. PMUs are to be distinguished from HPCs in that PMUs provide support to many different types of data which could be used to debug software performance whereas HPCs are special purpose registers that can be configured to count microarchitecture specific performance events. The set of possible events which HPCs may count are defined by the vendor and subject to change between processor generations. Several events (e.g., number of taken branches or L1 cache misses) are common across multiple CPU manufacturers.

The device class usually influences the capability of the PMU. For example, embedded processors typically cannot configure more than two HPCs simultaneously and typically have fewer than one hundred possible events that may be counted. The typical laptop, desktop, or server provides four configurable counters while maintaining two counters that always count CPU cycles and total retired instructions. Server class CPUs, such as those used in high performance computing and data centers may have several hundred more architecture-specific events that can be monitored when compared to laptop or desktop microprocessors.

The data collected by hardware sensors can be used to establish covert communication channels or for side channel analysis. This property of hardware sensors poses both risks and opportunities for information security. In research, risks posed by side channels are often of interest. For example, CPU power management sensor data have been used to leak Advanced Encryption Standard (AES) keys [19]. If power management sensors are strong enough to leak cryptographic keys, it is reasonable to believe that those same side channels might also be used to create robust application fingerprints. The data collected by PMUs are subject to similar risk-reward trade-offs. Controlling access to PMU sensor data is a key component of PMU-based machine learning research.

C. PERFORMANCE COUNTERS AS SENSORS

HPCs are unique when compared with other kinds of hardware sensors because they preserve knowledge about the internal state of the CPU–which is directly related to the code and data executing on the CPU. Recent HPC research has used the relationship between HPC data and program code to hypothesize that HPC events can represent nodes in a program's CFG [17]. The CFG is a directed graph containing all the legal states and transitions between states that can occur legitimately in a program. State transitions are manifested as function calls, branch instructions, or jumps. The instructions executed between state transitions are treated as a single node within the graph.

Basu *et al.* [17] presented a statistical framework for calculating the probability that two distinct CFGs exist with matching sequences of tuples for two, three, or four simultaneously collected events. The following list summarizes Basu's findings:

- If HPC events are precisely recorded every CPU clock cycle, then a counter of branch instructions and branches taken can represent the state transitions in the CFG.

TABLE 1. Commonly used tools to configure HPCs.

Tool	Strength	Weakness
Inline assembly	Minimal Overhead	Not portable
`perf`	Easy to use	Command line utility
`perf_event_open`	O/S provided & flexible	Difficult to use
OProfile	Post-analysis	Limited support
PAPI	High performance computing uses	Userspace only
libpfm4	Library to maps events to config	Requires other tools
VTune	Built by Intel for Intel	GUI-based analysis

- As sequences of event counts grow longer (i.e., the CFG grows deeper), the probability of finding a CFG with matching event tuples at each timestep decreases.
- Even though real systems cannot record HPC data every cycle, the probability of finding a matching CFG when events are counted over several hundred cycles is low enough to be useful in practice.

Basu conducted his analysis using a compiler-generated CFG, controlling program input, assuming a single instruction is executed per clock cycle, and nondeterminism in HPC values less than 5%. Basu's work provides a significant motivation for continued HPC research. If HPCs can accurately represent a CFG, there are potentially significant ways HPC data could enhance current software-based security mechanisms. A limitation of Basu's assumptions is that microprocessors are often capable of executing multiple instructions per cycle and the variation in data sequences over time is not well described. The operating conditions of real systems could inhibit the practical application of Basu's work; data collected under real operating conditions deserves further study.

D. COLLECTING HPC DATA

At the lowest level, HPCs are configured by writing a configuration bit pattern to a special purpose register. On x86-based systems, this is accomplished with the write model specific register (`wrmsr`) instruction. Similarly, the POWER Instruction Set Architecture (ISA) defines the move to special purpose register instruction (`mtspr`) to perform this operation. Instruction-level configuration provides a high level of flexibility but is difficult to implement in practice because the configuration details are often architecture specific and not easily interpreted by humans. Due to these complexities, researchers typically rely on a combination of tools to achieve research goals. This section describes common tools used in research, leaving fine-grained choices for tool configuration as a framework contribution described in Section IV.

Table 1 enumerates commonly used research tools and summarizes the primary strength and weakness. Understanding these tools and how they function helps in making research experiment design decisions. Across all tools, there are generally two different ways to access HPC data. Polling is a low-overhead, easy-to-configure approach that simply reads the values stored in the event counter registers. Sampling is an event-driven approach that records the counter values each time an event threshold is reached.

Inline assembly is the most direct and low-level approach to configuring and reading HPC values. When used in this way, users must be aware that HPCs are not saved and restored as part of a process-context switch. Therefore, the implementation must provide kernel-level modifications to save and restore HPC state during context switches [20]. Depending on the use case for HPCs, not saving and restoring counter values could result in significant variation. Inline assembly is a tool for directly studying event counter behavior, especially when only a single processor or a small subset of events is of interest. The task of writing correct inline assembly code for all possible event counter settings is difficult and benefits from layers of abstraction.

`perf` is a command line system tool that is provided as part of the mainstream Linux kernel. The two sub commands most commonly used to collect data are `perf stat` and `perf record`. Both sub commands provide command line interfaces to specify events to monitor and parameters to control data collection behavior. `perf stat` implements the polling approach to collect data. `perf record` implements a sampling approach to data gathering and records the instruction pointer address every time an event threshold is reached. By recording the instruction pointer each time a threshold is reached, a profile is developed to identify the code area where the overflow occurred. `perf` is powerful, but the command line interface makes it difficult to target specific portions of a program when collecting labeled data.

The `perf` tool is a command line wrapper for the `perf_event_open` Linux system call. Using the system call directly increases the flexibility significantly. Specific code can be instrumented and custom data output formats can be used to store data in formats that are easily consumed by automated tools, such as those tools that support a machine learning pipeline. Since `perf_event_open` is an operating system provided interface, the complexities of correctly handling context switches are handled automatically through virtualization of the HPCs [21]. The main drawback of using `perf_event_open` directly is the complexity of the interface, which provides the complete set of functionality through a single function. Research related to exhaustive testing of the Linux `perf_event_open` interface provides a rich source of examples to adapt to a specific use case [22].

OProfile is a command line utility that wraps the `perf_event_open` interface and offers similar features to `perf`. OProfile provides some useful post-analysis features such as reconstructing a call graph from sampled data. Similar to `perf`, OProfile makes it difficult to isolate specific functions or program areas independent of the main program. OProfile tries to map a common set of events to several processors and does not appear to offer a convenient way to select architecture-specific events.

The Performance Application Program Interface (PAPI) is a user space library designed to allow precise instrumentation of source code and was designed for high performance computing environments. PAPI maintains a strong balance

between portability and ease-of-use. The main drawback to PAPI is that PAPI is designed to monitor code running in user mode only.

`libpfm4` is a library with the sole purpose of translating text-based event names to proper raw specific architecture encodings or operating system encodings. `libpfm4` is used by PAPI to translate event names and provides a useful tool to configure HPCs when the `perf_event_open` interface is used directly.

The Intel-provided VTune software provides a robust Graphical User Interface (GUI) to diagnose application performance on Linux and Windows systems. Intel's understanding of their own microarchitecture and unique processor features makes VTune an important tool for analyzing applications that run on Intel platforms. The primary limitations with VTune are that the tool is designed for Intel products and does not support processors from other vendors. Additionally, as a GUI tool, automating the capture and storing of raw data is difficult. Command line tools support VTune's GUI features but are subject to similar data processing limitations as `perf`.

E. CACHE SIDE CHANNELS

In information security, a side channel is an unintended communication channel that an adversary can use to learn private information. Many side channels exist in computer systems, and researchers have identified several side channels in the caching hierarchy used by modern CPUs [23]–[26]. The description of a cache side channel provided in this section is not comprehensive and ignores many implementation complexities.

Recalling the discussion of Figure 1, caches exist to create a user experience of accessing a large and fast memory. Cache and main memory are also shared between the operating system and user programs simultaneously. Emptying the contents of a cache (e.g., flushing the cache) each time a different task executes results in significant performance penalties waiting for the data to be repopulated. Consequently, caches simultaneously store data from multiple tasks which might have different permission levels. In light of recent transient execution vulnerabilities, operating systems may choose to incur the penalty associated with flushing or invalidating cache lines on context switches to reduce the possibility of timing side channels.

Figure 2 presents a simplified view of caches as a function that maps a large set of possible addresses in main memory to a much smaller set of addresses in the cache nearest to the CPU. Given the disparity in address space size, it is trivial to choose two main memory addresses that would map to the same address in a particular level of cache (e.g., L1 in Figure 2). Consider two programs, *A* and *B* that run simultaneously on the same CPU core. Assume a precise timing source exists to distinguish between read requests when data is or is not in the cache. If Program *A* runs first, then Program *A* need only access data at 2^{15} addresses to completely populate the cache. After populating

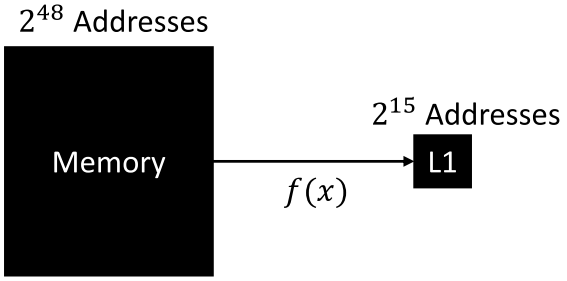


FIGURE 2. Function mapping memory addresses to cache addresses.

the cache, Program *B* runs and only reads some addresses with equivalent mappings in the L1 cache. When Program *A* is given time on the CPU, Program *A* can iterate through 2^{15} sequential memory reads to measure access time and identify a set of possible addresses used by Program *B*. This is the basis of a timing-based cache side channel because ideally, Program *A* should not be able to learn any information about Program *B*.

Cache attacks on real systems are more complex for several reasons. First, data is read in blocks rather than single addresses to benefit from the principle of spatial locality. Second, most caches allow for associativity, which means that a single main memory address can be mapped to more than one cache address at each cache level. Third, each CPU core typically has its own L1 instruction and data cache requiring an attacker and victim process to reside on the same core. Attacks against L2/L3 caches are possible but tend to have much slower read rates. Lastly, caches may choose different policies (e.g., Least Recently Used or First In First Out) to determine which data is kept in the cache. Despite these challenges, timing-based cache side channels are achievable on most modern CPUs [1], [3], [4].

PMUs provide HPCs as a sensor to monitor cache hits, misses, flushes, and several other events related to cache performance. Cache thrashing happens when data is evicted from the cache and subsequently needed. Cache thrashing causes undesirable performance degradation that typically is not present in legitimate programs. Cache side channel attacks tend to create significant shifts in cache behavior making the cache side channel detectable by monitoring HPCs. A limitation of this approach is that cache side channel read rates can be slowed down, causing cache misbehavior to be nearly indistinguishable from legitimate behavior.

F. SPECULATIVE AND TRANSIENT EXECUTION

The terms speculative and transient execution are closely related. Performance is a primary design consideration in modern microprocessors. Recall from Figure 1, the time to access data in main memory is 2-3 orders of magnitude slower that the CPU's speed. Many situations arise when the CPU can either decide the next action or wait for data to ensure the right action is always taken. The performance penalty of waiting for data is significant, so modern processors speculate

or predict the likely next action. Speculative execution refers to all code that is executed before the true data is known to confirm the prediction. Speculative execution encompasses both correct and incorrect predictions. Speculative execution can occur in code flows (e.g., branch prediction) or in data flows (e.g., cache prefetching).

Speculative execution encompasses all actions taken based on a prediction–even if the prediction was correct. Transient execution refers to executed instructions whose results are never committed. Transient execution never results in incorrect behavior from the user's perspective. However, transiently executed instructions cause state changes in the microprocessor. The cache is one example of a microarchitectural component whose state changes as a result of transiently executed instructions, but others also exist.

Prior to the evolution of transient execution attacks, cache side channels were mostly passive listening channels. That is, the adversary could only observe victim behavior, but could not control what victim data to read. In practice, the listen-only property of cache side channels reduced information security concerns. Transient execution attacks offer adversaries a mechanism to control what data is loaded into a microprocessor's cache to actively leak specific information.

Most PMUs provide HPC events related to a program's transient execution control flow. For example, the POWER9 provides the `PM_BR_PRED` event to count the total number of branches where hardware made a prediction about the branch outcome. Additionally, the POWER9 provides the `PM_BR_MPRED_CMPL` event to count the total number of mispredicted branches. By applying Basu's hypothesis to hardware events that reflect transient execution flow there might be opportunities to identify CFGs associated with speculation-based gadgets.

The information provided in this background section established a high-level perspective of the operating environment in which computer programs run. Basu's prior work [17] was summarized to provide a hypothesis-driven foundation to motivate research in this problem domain. Specifically, the limitations of Basu's simplifying assumptions and the complexities of modern computer hardware were discussed to establish a need to describe HPC sequence behavior on real systems.

III. RELATED WORK

This section describes related research that was not discussed in Section II. This section primarily focuses on information security papers, though HPC data has been used in automated performance analysis, debugging, general high purpose computing, operating system support, and power analysis [20]. An excellent summary of non-security and security-focused HPC papers is available in [20]. The related work is discussed in the categories of HPC behavior characterization, hardware malware detectors (HMDs), exploit detection, and arguments in favor of or against the use of HPCs in information security applications.

A. HPC BEHAVIOR CHARACTERIZATION

Early HPC research explored the trustworthiness of data gathered using HPCs. Weaver used an empirical approach to measure the variation when standard benchmark programs executed [27]. Weaver observed that HPC variation could be reduced to reliable levels, but there were nuances associated with correctly configuring event counters [27]. In a later work, Weaver *et al.* [28] further examined the determinism of events on multiple processor architectures and found that few events are deterministic, which increases the difficulty of using HPC data for inexperienced users. More recently, Weaver implemented an extensive test suite and system call fuzzing utility for the `perf_event_open` interface [22] which provides useful technical details for new tool implementations that abstract the complexities of the system call.

B. HARDWARE MALWARE DETECTION (HMD)

HMD research focuses on identifying software executing without the user's permission or intent (malware) rather than the mechanism the adversary uses to achieve an objective. Researchers have proposed several kinds of malware detection systems using microarchitectural data. As early as 2013, well before Spectre and Meltdown, malware detection was proposed using HPC data to classify malware, rootkits, and hardware design considerations to enable secure hardware-based malware detection systems [29]. Others used similar techniques to develop HPC-based detection systems for mobile devices [30] and improving rootkit detection [31]. The most common approach for malware detection uses supervised learning, where data used to develop a model is labeled as belonging to the malware or benignware class and statistical relationships are estimated to predict class membership for the new data. Some HPC research uses unsupervised learning techniques and includes analysis to determine which HPCs are suitable to detect malware [32].

C. EXPLOIT DETECTION

Exploit detection is considered separately from HMD research because this research tends to focus narrowly on a specific attack technique. In 2015, Herath and Fogue [33] demonstrated the use of HPCs on the Intel x86/64 architecture to detect a variety of exploit techniques, including Return Oriented Programming (ROP) [34], Rowhammer [35], Rootkits, and cache side channels.

ROPSentry detects ROP attacks with data gathered from HPCs at low sampling rates (100 ms) [36]. The authors recognized practical limitations of machine learning systems using HPC data and focused their analysis on understanding ROP attack behavior to identify a policy threshold for various metrics. Threshold-based detection systems for different exploit techniques have been proposed by others [37], but are subject to the challenges of low-bandwidth attacks.

In one of the earliest Spectre and Meltdown detection publications, WHISPER was proposed to perform

run-time detection of cache side channel attacks and Spectre/Meltdown with greater than 99% accuracy. WHISPER used both PAPI and `perf` to collect data [10]. Attack detection was shown to be effective at coarse sampling rates (100 ms) and incorporated application-event based sampling, such as sampling after n encrypted bits were generated by an encryption algorithm, rather than a fixed time interval.

The time series nature of HPC event data has been the focus of research for detecting transient execution attacks. Alam proposed a two-part detection system composed of a correlation module and a classification module [38]. The correlation module used dynamic time warping to determine the similarity of two sequences, while the classification module performed classification using a variety of machine learning techniques. Alam's research gathered data at sampling intervals of 10 ms and 100 ms using the `perf stat` tool [38].

Cache side channels tend to be considered in transient execution attack detection systems. However, Wang makes the important observation that caches are not the only side channel possible when instructions execute transiently [12]. Rather than focusing on a specific attack, Wang generated thousands of attack variants and designed a detection mechanism for the critical paths required to load data into any possible side channel.

D. ARGUMENTS FOR AND AGAINST HPC DATA

The arguments in favor of HPC data for attack detection are both theoretical and practical. In the development of a theoretical foundation for program integrity checking, Malone *et al.* hypothesized that programs can be approximated as a mathematical relationship between event counts such that by running a program multiple times with varying inputs the performance counters could be used to identify the relationship between the executed program and the observed event counts [16]. Malone *et al.* measured user-space event counts for retired instruction, branch, floating point events, L1 cache writes, and completed Input/Output (I/O) operations by modifying the `perf_event` system call to read data every 10,000 cycles [16]. Basu's statistical analysis to estimate the probability of creating a matching CFG [17] supports Malone's hypothesis. HPCs are not the only source of data provided by PMUs. Tice *et al.* [39] proposed using branch tracing facilities, a common capability implemented in PMUs, as a mechanism to detect Control Flow Integrity (CFI) [40] violations.

In a survey of HPC uses across multiple domains, Das observed that all 41 information security papers chosen recommended HPCs, while only about fifty percent of non-information security papers recommended HPCs [20]. Some security research expresses a strong disagreement with the notion that there is a causal relationship between high-level software and low-level hardware events [15]. The contrary work attributes research success with HPCs to comparatively unrealistic experiment design and uses low classification accuracy rates ($< 80\%$) for non-deep learning models as a foundation for argument [14].

IV. FRAMEWORK

The information presented in Section II and III provided motivation, examples of success, and areas of concern in HPC research. This framework aims to address concerns expressed by Weaver and McKee [27] that HPC configuration is difficult and Das *et al.* [20] that research exposes too few details about how data is gathered to create reproducible results. Research across multiple problem domains, has yielded promising results using HPC data and the framework proposed in this section draws several concepts together to further HPC research.

The elements of this framework are summarized in Figure 3. This section introduces the framework, beginning with the formulation of a model for the microarchitectural state as a dynamic system that changes over time. Following the model description is an overview of the classes of problems for which HPC data has been applied to propose a solution. The next section enumerates technical considerations (factors) and settings (levels) for configuring HPCs. The technical considerations and settings have nuanced relationships that are explained in greater detail. The remainder of the framework steps are presented in Sections V-VII. Technical considerations and settings are used as the foundation to design a data collection architecture in Section V and evaluation criteria in Section VI.

A. THE MODEL OF MICROARCHITECTURAL STATE

There exists a complex interaction between the full set of executing programs, the operating system scheduler, and system resources. This section leverages a classical dynamic system model to describe factors which must be considered when gathering data to build goal-driven systems using HPCs.

Equation 1 describes a typical programmer's view of using a function to implement a task. The programmer assembles a sequence of instructions to compose f so that input x produces the output y. Except for real-time systems, the programmer's view of correctness ignores the time component. A programmer might find time relevant with regard to algorithmic complexity (e.g., ''Big O'') but generally does not consider the number of clock cycles to execute logic when defining a program's correctness. Equation 1 is a simplified view of computer programs because legitimate programs exist that do not conform to the mathematical definition of functions.

$$f(x) \rightarrow y \tag{1}$$

Architectural logic and microarchitectural state are used to execute instructions at a layer below the program logic. The architectural logic and state are influenced by time and nondeterministic events. The representation of a CPU's microarchitectural state may be thought of as a classical dynamic system [41], as shown in Equation 2. For every

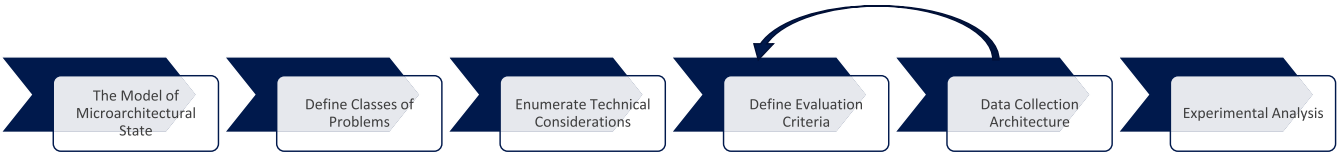


FIGURE 3. Framework and methodology overview.

function f that a programmer defines and executes, there is a function internal to the microprocessor m operating on the microarchitectural state s present at time t. The programmer's function f terminates at some time t when the microarchitectural state has committed all operations required to produce y correctly. Under this definition, the function f depends on the microarchitectural state and the input value x.

$$m(s^{(t-1)}, f, x) \rightarrow s^{(t)} \tag{2}$$

An obvious and simple point must be made with regard to Equation 2. To develop precise models, some knowledge about f and the data x must be known. In current CPU architectures obtaining this granularity of detail at run-time is difficult. Additionally, even CFG-based HPC research often neglects the influence x has on the sequences of states traversed in a program's graph. In the absence of representations for x, practitioners should strive to select HPCs that preserve some information about x.

Although Equation 2 incorporates time and the dependence of the current state on a previous state, the model is still idealistic. In reality, the state at time t includes the results of many other signals from the operating system and internal to the CPU. Therefore, the formalization of the microarchitectural system state is proposed in Equation 3. $s'^{(t)}$ represents the actual microarchitectural state at time t. A limitation of $s'^{(t)}$ is that it is only partially observable in real systems. Visibility of the microarchitectural state is limited to at most four out of several hundred possible events related to microarchitectural state. HPC data are single-point measurements of microarchitectural state changes summarizing the number of events in a period of time. A 5 μsec sampling interval on a 3GHz processor will summarize the events during 15,000 timesteps.

The function $g^{(t)}$ is introduced to the model to account for the complex external signals that influence the processor's microarchitectural state. The external signals, represented as g at time t, could include things like asynchronous interrupts, operating system scheduler decisions, system load, changes to process priority, cache evictions, detection of incorrectly predicted branches, and several others. For practical purposes, $g^{(t)}$ cannot be fully known or modeled, but the influence exerted by $g^{(t)}$ on the microarchitectural state cannot be ignored. HPC research should include a detailed experiment design plan to create conditions where variation exists in the data collected for model development.

$$m'(s'^{(t-1)}, g^{(t)}, f, x) \rightarrow s'^{(t)} \tag{3}$$

TABLE 2. Consolidated HPC goals adapted from [20].

Category	Use Case
Performance	Optimization
	Debugging
	High Performance Computing
	Support to Operating Systems
	Power Analysis
	Performance Analysis
Security	Exploit Technique Detection
	Side Channel Attack Detection
	General Purpose Malware Detection
	Firmware Verification
	Control Flow Integrity
	Virtual Machine Introspection
	Vulnerability Analysis
	Cloud Services Side Channel Detection

B. DEFINE CLASS OF PROBLEM

Mindful of Equation 3, the next step is to identify the problem class for which HPCs are being used. Defining the problem class eases the search for related work and facilitates informed decision making during experiment design. It is important to recognize that there are many legitimate goals for HPC data. At a high level, these goals are either performance-based or security-based. The survey conducted by Das *et al.* [20] presents an excellent summary of several HPC papers across multiple problem areas. A summary of problem classes is provided in Table 2.

When an information security application is the primary goal, a threat model describing adversary capabilities should also be defined so that the suitability of the data collection architecture can be evaluated. Threat models are specific to a particular problem and because this research focuses on a general approach, these details are omitted. Future research may consider how aspects of the proposed data collection architecture could be replaced while maintaining flexibility to collect labeled data and strong security properties.

C. ENUMERATE TECHNICAL CONSIDERATIONS

In order to conduct repeatable HPC research, there are several technical considerations to explicitly state. This research focuses on factors specific to HPCs, though a similar organization could be performed for other data sources provided by PMUs. Table 3, arguably one of the principle contributions of this work, provides an overview of the considerations discussed in this section. Experiment factors are a unique combination of settings or levels required to support a research question, while all other factors are design decisions that guide the use of existing tools or require the

TABLE 3. Overview of technical considerations and design decisions.

Factors	Levels	
Collection Technique	Poll	
	Sample(Random Instruction)	Skid Data to Examine # of Pages for Buffer
Measurement Interval	Event-based Time-based	
Permission Level	User Mode Kernel Mode Hypervisor Mode	
Collection Aperture	System-wide Core(s)	
	Process or Thread	Include Children Exclude Children
Chosen Events	Relationship to Behavior	On-Core Events Off-Core Events
	Hardware Limitations	Counters Always On Multiplex Counters
Monitoring Context	Unknown Code	
	Known Code	Known Location Unknown Location
Data Requirements	Learning Mode	Supervised Unsupervised Semisupervised
	Problem Type	Classification Regression
	Data Format	Sequences Feature Vector
Experiment Settings	Various	

development of a new tool. Future researchers may wish to use the template provided in Table 3 to concisely express the configuration of HPC data gathering tools.

Collection Technique

There are two ways to collect or read HPC data. The first way is the polling technique. In the polling technique HPC values are read directly and the only information recorded with the poll is the value of the counter(s). When hardware-based instructions are used to poll counters, the event counters are accessed with individual read requests similar to the way general purpose registers are accessed. Some abstractions, such as `perf_event`, provide the individual read instructions and allow organizing counters into groups so the data for multiple HPCs can be processed in software through a single `read` system call. Polling is used to create a vector with N features, where N is the number of configured events captured in a period of time. Polling also can occur at scheduled intervals to create time series sequences with N features per timestep. Unless polling can be implemented with dedicated hardware, the time series sequences will vary due to dynamic frequency scaling, system load, and the operating system scheduler decisions.

The second data collection technique is sampling. PMUs provide multiple options to configure sampling. For example, the IBM POWER9 offers continuous sampling, random instruction sampling, and random event sampling [42]. Random instruction sampling creates PMU interrupts based on an overflow condition. The overflow condition can approximate time-based samples (e.g., interrupt every c cycles) or can be specific to a behavior of interest (e.g., interrupt every b branch mispredictions). A benefit of the sampling approach is that contextual data can be accessed, including the contents of general purpose registers and the instruction pointer. Obviously, choosing to collect more data will have an adverse effect on the time to record a sample. Skid is a parameter to request a limitation in the number of instructions that execute between an event occurring and the operating system kernel being able to record the event. This is a request, not a guarantee, and the ability to influence this behavior might differ between processor architectures. Requesting minimal skid comes with a performance penalty because deep execution pipelines must be halted abruptly. Conversely, increased skid contributes to data variation because a larger amount of unknown pipeline execution is allowed to occur.

Measurement Interval

When time series data is of interest, there are two approaches to establishing the collection interval or collection rate. In the first approach, the polling method is used to schedule HPC reads after t time has elapsed. This approach is subject to the limitations of the operating system scheduler, which is constrained by a minimum execution time before context switches occur. If the logic of Basu's framework [17] is followed, then faster collection rates are desirable.

The second approach schedules an interrupt after c cycles and records the measurements of counter values in an interrupt handler. Although a small value for c (e.g., $c = 100$) could be chosen, doing so is impractical because of significant performance penalties due to continuous interrupts and the speed of modern CPUs.

Permission Level

Code may execute with hypervisor, kernel, or usermode privileges. The purpose for building a system using HPC data is the primary consideration. A cloud service provider constructing a detection system for misbehaving hosts leaking information from co-resident virtual machines [43] may choose to focus on data collected by code running at the hypervisor permission level. When CFG-based code identity is of interest, researchers might collect usermode data only. Any combination of permission levels is possible and requires use case analysis or experimentation to determine the best settings. Users should understand that trying to collect data for a specific privilege level may not be precise because information about instruction privilege level is not always available when an event is counted. This subtle property could have negative consequences for information security use cases.

Collection Aperture

The term *collection aperture* is used to define the visibility scope for collected data. The system-wide aperture

aggregates HPC counts for all processes or a chosen subset of processes. The process or thread-specific aperture collects data for a unique process and optionally the process's children regardless of the core a process executes on. The core aperture identifies a specific core or set of cores that are of interest.

Systems may implement a combination of apertures to best address the problem being solved. Aperture is an important consideration in the selection of a tool because not all tools provide adequate support for the desired aperture. For example, Das *et al.* [20] uncovered how a process aperture could negatively affect the accuracy of data when PMUs were configured with assembly instructions due to HPCs not being saved and restored during operating system context switches.

Chosen Events

Choosing the correct set of events for a particular problem is difficult, with many opportunities for future work. This framework section does not exhaustively explore techniques for event selection, but establishes some guiding principles and commonly used approaches. Researchers should articulate a methodology or rationale for a specific decision.

One aspect of choosing events is hardware limitations. PMUs generally provide two read-only HPCs for counting retired instructions and CPU cycles plus four configurable counters to suit user needs. Vendors impose different limitations about which event counters may be configured simultaneously. The ''counters always on'' level refers to the condition when the number of events counted does not exceed the hardware limitations.

Software might optionally be used to multiplex HPCs. Multiplexing HPCs allows the number of events being monitored to exceed the limitations of the number HPC registers by rotating which events are active during a period of execution. Multiplexing is invaluable from a performance analysis perspective because it increases the visibility into possible performance bottlenecks. Conversely, multiplexing is generally considered undesirable when continuity of information is important. Information security is an example where visibility gaps in the data stream could have consequences. As more events are monitored, the possibility of missing data related to an attack increases. Losing data is an inherent trade-off of the multiplexing approach but there are benefits of gaining visibility into multiple areas of a microarchitecture. Although most information security research restricts the number of events to the number of available HPC registers provided by hardware, performance analysis use cases might have strong reasons to multiplex events.

Another aspect of choosing events that research considers is the small super set of events common across multiple CPUs and with widely supported utilities like `perf` to make solutions portable across multiple CPU architectures. However, this may not always yield the event set most correlated to a phenomenon. Modern processors, especially server-grade processors, can have hundreds of candidate events for task-specific on-core events (e.g., L1 cache hits) or off-core events that occur with shared input-output resources that might not easily be correlated with a specific task context [44]. Off-core events can be layered for use in distributed systems and are outside the scope of this framework but should be considered for future extensions.

There are two basic approaches to selecting which events to monitor. The first approach selects events using prior research that produced good results. Several published papers for Spectre, Meltdown, and various cache attacks [9]–[11] use correct or incorrect branch prediction events and cache hit sequences because these types of events are expected to have high correlation with speculative execution and cache side channels. Others, such as Malone *et al.* [16] and Basu *et al.* [17], identify a small event set based on a theoretical view of HPCs as a representation of a CFG. Malone suggested that the total number of retired instructions, branch instructions, floating point instructions, return and call instructions, I/O operations, and L1 cache writes form a good representation of a program's CFG [16]. Basu's framework focused on the number of retired instructions, total branches, taken branches, and integer instructions [17].

The second approach to selecting events uses experiments to evaluate HPC data against the criteria of consistency and distinctiveness. The consistency and distinctiveness criteria can be used to study which HPCs are likely to be associated with new behaviors of interest. Machine learning tools can be used on small amounts of experimental data to support HPC selection prior to developing large experiments for training neural networks. For example, unsupervised dimensionality reduction approaches such as t-SNE or principal components analysis [41] could be used to assess whether the data gathered by an HPC configuration forms clusters when a specific property under investigation is present in the system. Similarly, supervised classification tools such as Random Forest Classifiers (RFCs) [45] could identify HPCs associated with a decision boundary. Some RFC implementations expose the calculated relative feature importance which would facilitate determining which data contributes to the decision boundary.

There are good reasons to investigate architecture-specific HPCs for different problem conditions. For example, the IBM POWER9 counts events related to a proprietary data address and recirculation queue (DARQ) that issues load and store operations to cache units [42], [46]. In a cache side channel analysis use case, it is possible that this type of architecture-specific event counter provides meaningful data.

A desirable quality for microarchitectural events used to develop various types of models is consistency. For the same program input and execution conditions, event sequences should be consistent (i.e., visually similar when plotted as a time series sequence). Inconsistent counters will confound the characterization of the CFG from HPC event sequences, weakening the fingerprinting properties. Recalling Equation 3, inconsistent counters might provide some information about external signals, $g^{(t)}$, but

requires further study to determine the event's relationship to $g^{(t)}$.

Events can have consistent behavior but yield limited distinction among classes. The number of retired instructions at each timestep during a program's execution could be consistent, but if a second program produces the same sequence then the number of retired instructions is not useful for distinguishing classes. The search for distinct events uncovers events that exhibit responses when chosen execution conditions change. Even with painstaking analysis, it is likely that the chosen events will be based on empirical evidence and professional judgment.

Monitoring Context

Monitoring context refers to the data collection approach's awareness of location or context within executing code. The system-wide collection aperture used with the `perf stat` command to collect sequences does not include any filtering to identify specific functions executing when data is collected. In contrast, the system-wide collection aperture with `perf record` logs the instruction pointer to capture the approximate location in a running program at the moment a sample is recorded. Monitoring context will be chosen differently for different research questions or goals. For example, low-overhead cache side channel detection systems are not likely to care about which code causes a malicious cache access pattern to take place. However, researchers confirming the identity of a CFG using HPC events will take care to collect sequences where the location is precisely known and labeled.

Data Requirements

The analysis tools applied to the collected data are an important factor. There are different approaches to machine learning with different data collection requirements. Three broad categories of machine learning are described in Table 3. Supervised learning finds statistical relationships between input features and a known response variable. The response could be a point value or label. Unsupervised learning does not use a response variable but tries to find statistical relationships between differing groups of features [45]. Semisupervised learning uses a combination of unsupervised and supervised approaches. Classification problems predict which labeled class a previously unseen set of data belongs to, while regression problems predict a point value. HPCs can be used to collect sequences or point values. When sequence-based learning techniques are used, such as Long Short Term Memory (LSTM) [41], training time and sequence length limitations should be considered because of the influence on collection interval decisions.

Experiment Settings

The experiment settings aspect is intentionally vague to allow researchers to define factors and levels associated with new research questions. Experiment settings could be anything that is related to a researcher's goal. Some settings include factors like system load, CPU architecture, programs to fingerprint, or the microarchitectural approach.

An important point to consider when designing experiments that will use machine learning techniques is that the data generated by the collection process introduces bias. The bias-variance trade-off [45] is a fundamental tension in machine learning. For example, a model trained with data on only one CPU microarchitecture will become biased to recognize patterns present on a single chosen microarchitecture. If multiple types of CPUs exist in production systems, data collection should address this challenge. This challenge can manifest itself in security research when data is collected using known attack variants when researchers do not consider how adversaries might adjust an attack to create new variants.

V. DATA COLLECTION ARCHITECTURE

This section describes a framework implementation that simplifies the task of gathering labeled microarchitectural data suitable for machine learning model development or statistical analysis. Figure 3 depicts the framework as a sequential process, recognizing that the development of evaluation criteria and a data collection architecture includes feedback. New technical considerations may arise when implementing the data collection architecture that require new evaluation criteria. Reflecting on evaluation criteria and data collection architecture independently while incorporating feedback provides a more substantive understanding jointly. The data collection architecture is presented first so that the evaluation criteria can be understood in the context of the architecture.

A. DESIGN GOALS

Section II described many tools which may produce data for machine learning applications. Before building a new tool, it is important to establish why it is necessary. The three design goals proposed in this section address gaps in the current set of tools used in HPC research.

1) EASY TO CONFIGURE AND USE

There are many excellent tools for performance analysis that make use of HPC data. Low-level tools, such as the `perf_event` interface can be daunting to implement. The data collection architecture should make the researcher's task of selecting settings from Table 3 simple. Researchers should focus on performance events of interest and experiment design rather than correctly using an obscure system interface or scraping data from command line tools.

2) EXPERIMENT DRIVEN

Deliberate experimentation should drive research. The data collection architecture must provide flexibility under a wide variety of experimental conditions. Researchers should be able to obtain fine-grained, known program location when collecting data, and experiment settings should be traceable to the specific experiment run that produced the data.

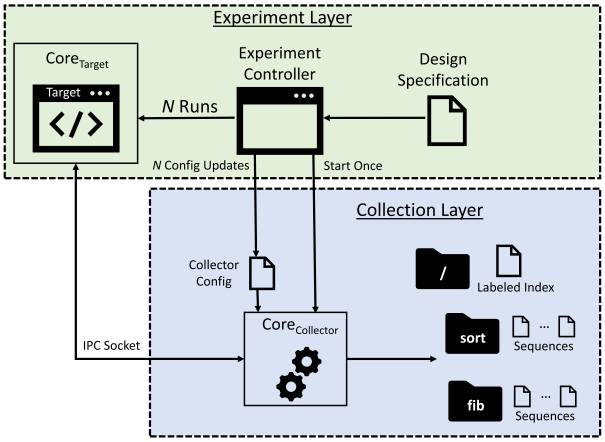


FIGURE 4. Data collection architecture.

3) TIME SERIES ANALYSIS APPROPRIATE

The benefits of obtaining time series data, resulting from the real execution of real programs, are potentially significant. Unlike fixed-length feature vectors, time series data collection produces variable length sequences that present organizational and storage challenges. The data collection architecture should support time series data that is appropriate for analysis with time series techniques, while minimizing the data wrangling performed by researchers.

B. IMPLEMENTED ARCHITECTURE

The data collection architecture is shown in Figure 4. The implementation of the framework provides support for the known code and known location on a single system. However, the approach is general enough to enable a system-wide collection aperture because the request-response protocol can be used to initiate monitoring sessions without instrumenting code to obtain system-wide or process-specific data. There are two key benefits enabled by a client-server approach to collecting HPC data. First, consider the microarchitectural state model described in Section IV. System-wide and even code-specific models implemented using tools like PAPI do not provide a mechanism to store information about the input data x. The client-server model presented in this approach could be extended to enable transmission of x or summary properties about x to enable context-aware decisions. Second, the client-server approach could be extended to gather data when code executes in a distributed environment. These two benefits are outside the scope of this research, but are viable future HPC research areas.

The architecture uses Interprocess Communication (IPC) to establish the client-server channel that allows the client to provide context information to the server, which the server uses alongside a settings file to configure HPCs. Session management refers to the process of configuring HPCs (i.e., session establishment), clearing HPC configurations (i.e., session tear down), and storing captured data to disk. The time penalty of session management is viewed as irreducible. The client-server approach can be easily adapted for online inference tasks so that control flow in sensitive processes can be halted when unexpected behavior is encountered. In addition to enabling online inference, the client-server model allows for collecting HPC data from precise locations in the original source code.

Separating concerns between data collection and experiment design eases configuration and promotes experiment-driven analysis. Data collection parameters are stored in a key-value format in a configuration file that is read each time a new collection request is received by the collector. For production systems, when the data collection parameters are known, execution overhead could be reduced by injecting the configuration data using system environment variables or memory mapped locations.

The experiment design layer is user-specified, meaning that the user identifies experiment factors and levels, then specifies the run sequences. The experiment design layer writes the configuration information prior to each run where data will be collected. Additionally, the user leverages the experiment control layer to configure experiment-unique factors (e.g., implement chosen type of system load) and launch the target application to collect data from. This research implemented experiment control using `bash`, but using other languages (e.g., `python`) is feasible. For each experiment (e.g., the complete set of runs), the collector is started once and the experiment controller communicates with the collector by writing the collector configuration and data labels to a file.

To provide time series data support, the collector maintains an index to match run metadata with pointers to individual run sequences. The index file approach allows for flexible storage of metadata (e.g., run factors/levels, collection interval, etc.) and multiple labeling for sequences.

VI. EVALUATION METHODOLOGY

Section IV presented a general framework for HPC research while Section V described one possible implementation based on the framework. This section discusses the evaluation methodology using the framework implementation to characterize the behavior of the data collection process and the approach's performance overhead. The four evaluation criteria begin in Subsection VI-A. Subsection VI-E describes two data collection experiments to gather data to apply the evaluation criteria. Subsection VI-F describes two techniques used to automate outlier detection and remediation. Lastly, Subsection VI-G describes the three computer systems used to execute the data collection experiments. The evaluation criteria results are presented in Section VII.

A. CRITERIA #1: STATIONARITY OF ELAPSED TIME

In time series analysis, stationary models maintain an equilibrium with constant variance about a fixed mean [47]. Although data collection tools are imperfect, the values observed in the time domain should be stationary. A non-stationary time domain would indicate the need for remedial measures when classical time series analysis techniques are

used and might be a source of irreducible noise with negative effects on deep learning models.

A unit root is a parameter in time series models that accounts for a shift in mean or variance as time changes. Stationary data does not have a unit root. The HPC collector implementation should produce stationary data without a unit root. The augmented Dickey-Fuller (ADF) is a statistical test used to test for the presence of a unit root in the sequence [47]. The formal hypothesis for the ADF test is shown in Equation 4. The number of instances where H_0 is not rejected are counted to determine if nonstationarity occurs as a result of the data collection process. This test is performed on a quiescent system with minimal operating system load. The ADF test is appropriate for sequences of elapsed time measurements, but other sequences of event counts might not be stationary, depending on the relationship of the event to the running program.

$$\begin{aligned} H_0 &: \text{There is a unit root in the series} \\ H_a &: \text{The time series is stationary} \end{aligned} \tag{4}$$

Lastly, the median value for the elapsed time is estimated for the different measurement techniques at various collection intervals. When the median value for the elapsed time measurement is obviously greater than the specified collection interval then the chosen collection interval is too small for the architecture to handle reliably.

B. CRITERIA #2: DATA LOSS

Time series data collection tools for HPCs should not lose data. This proposed evaluation criteria measures lost data by counting events that are invariant between program runs when given the same input. The total number of retired branch instructions is considered invariant because even if a process is interrupted and a context switch occurs at arbitrary points in time during a program's execution, the aggregated retired branch instructions should be constant. A subset of data when only the branch instructions event is enabled (see Table 4) is used to measure data loss for each collection technique at various rates for two different programs. Users applying this framework can monitor event counts over repeated program runs and use variation in event counts to help determine if the chosen configuration might be discarding data.

The standard deviation of the total branch instructions between runs for an ideal collection technique would have a standard deviation near zero because the total number of branch instructions would never change. Bootstrap resampling is used to estimate the 95% confidence interval for the standard deviation of the total number of branch instructions. The confidence intervals can be used to determine the possible challenges associated with data loss in the collection techniques.

C. CRITERIA #3: SEQUENCE VARIATIONS

Experimental data is collected to evaluate the consistency of invariant events for a program between runs. Two different approaches are applied to compare results. The first method uses the Kolmogorov-Smirnov test (KS test) [48] at a type I error rate of 0.05 ($\alpha = 0.05$) to compare distributions of branch instruction event count sequences from pairs of runs with identical factors. Equation 5 provides H_0 and H_a for the KS test. A pairwise comparison between observations was performed because there does not exist a standard sequence to measure each individual sequence against. An α adjustment was not performed to account for multiple comparisons because α would be quite small making it unlikely for statistical tests to reject H_0. At $\alpha = 0.05$ approximately 5% of the tests will reject H_0 when the sequences are not actually different. There are $\binom{100}{2}$ combinations of run pairs per treatment. These tests are efficient on modern computer hardware, so the KS test results are obtained across all pairs of runs. For each run, the sequence was split into run windows of n observations. For the smaller collection intervals (5 μsec and 10 μsec) $n = 20$ was used. The sequences for the 100 μsec collection interval are significantly shorter (typically no more than 12 observations per sequence), so $n = 5$ was chosen.

When H_0 is rejected, there is evidence that at least one of the data points in the window does not come from the same continuous population. The total number of failures-to-reject H_0 is counted over all windows and the ratio of failures-to-reject versus the total number of tests performed is computed. A dataset with $\binom{100}{2}$ values of the ratio is created and the data is resampled to estimate the confidence interval for the ratio on each of the experiment factor level combinations.

$$\begin{aligned} H_0 &: F(t) = G(t) \quad \forall t \\ H_a &: F(t) \neq G(t) \quad \text{for at least } 1t \end{aligned} \tag{5}$$

The second method to compare raw sequences is pairwise Dynamic Time Warping (DTW) using the calculated DTW measurement as a score for similarity between the two sequences. The DTW implementation in the `tslearn` [49] computes the optimal path to align two sequences and then calculates the Euclidean distance as shown in Equation 6. The two sequences being compared are represented as a and b while the value at the sequence index i indicates the number of executed branch instructions since the previous timestep. DTW is a well-established dynamic programming algorithm that originated in speech recognition research [50] and has been applied in cross-virtual machine side channel detection research [43]. A 95% confidence interval for the DTW measurement is calculated using the bootstrap method.

$$DTW_{measurement}(a, b) = \sqrt{\sum_{(i,j)\in\pi} \|a_i - b_j\|^2} \tag{6}$$

where π is the optimal alignment path [49]

D. CRITERIA #4: COLLECTION OVERHEAD

The developed framework implementation achieves code location aware HPC monitoring. Achieving this goal imposes an additional performance penalty not found in systems

configuring HPCs in the kernel on context switches or systems that gather data with a system-wide aperture. Therefore, two aspects are considered when measuring the performance overhead. First, the overhead associated with establishing a monitoring session should be measured. In contrast with data collection using the `perf` utility, the instrumented code must communicate with a collector to configure the HPCs. Second, the overhead to collect and store data after a session is established should be considered.

The first overhead aspect, session establishment, was estimated using data with the sampling mode set to "None" (e.g., data collection turned off) as the control group. When the duration factor is 0 ms, no work is performed, thus isolating the session management overhead. The difference between two randomly sampled data points, one from the control population and one from the treatment population (poll or sample) was calculated. A simulation with one million random samples was used to obtain a point estimate of the mean difference between the two populations, thus providing a single value to describe session establishment overhead in the average case.

The second overhead aspect, data collection and storage, was estimated by subtracting the point estimate for the mean time to establish a monitoring session from the total execution time at 10 ms, 1 sec, and 30 sec to estimate data collection overhead separate from management overhead. The result of subtraction can be negative. Negative values are an indication that the data collection mechanism imposes minimal overhead once the session establishment overhead is accounted for.

A program that executed integer operations in a loop established the session overhead baseline. To compare the proposed implementation's performance in a more realistic environment, four programs from the MiBench [51] microbenchmarking suite were instrumented. The execution time data collected with the framework implementation disabled was used as a control group to establish a baseline of execution times for the collection techniques. The slowdown ratio is calculated as shown in Equation 7.

$$\text{Slowdown} = \frac{\text{Execution Time}_{\text{collection on}}}{\text{Execution Time}_{\text{collection off}}} \tag{7}$$

E. TWO DATA COLLECTION EXPERIMENTS

Two data collection experiments were defined to support the four evaluation criteria from Section VI. The first data collection experiment provides data for the first three criteria, while the second data collection experiment gathers the necessary data to support the third criteria. The detailed framework settings are described in this section.

1) CRITERIA #1-3 DATA COLLECTION EXPERIMENT

In both sampling and polling techniques, the elapsed time between measurements is a request, not a hard requirement. Assuming a constant CPU operating frequency, collection tools that promote consistency will gather data with minimal variation in the time elapsed since the previous measurement.

TABLE 4. Data collection factors and levels for evaluation criteria 1, 2, and 3.

Factors	Levels	
Collection Technique	Poll	
	Sample	
Measurement Interval	Poll	Time-Based (5 μsec, 10 μsec, 100 μsec)
	Sample	Event-Based in Cycles (16K, 32K, 320K)
Chosen Events	Intel	branch-instructions, cache-misses, LLC-stores, ref-cycles
	IBM POWER9	branch-instructions, cache-misses, LLC-stores, L1-icache-loads
Experiment Settings	Program	bubblesort, fibonacci
	Number of Events	Choose 1, 2, 3, or 4 Events

A subset of technical considerations from Table 3 was used to identify factors and levels to collect data for analysis. Technical considerations and experiment-unique settings, which are not varied during an experiment are described as parameters with settings. Table 4 shows the factors and levels while Table 5 shows the parameters and settings. For each combination of factors and levels 100 observations are obtained. The number of events and the event set chosen is treated in the analysis as a nuisance factor which might contribute some effect, but is not a focus of the analysis.

2) CRITERIA #4 DATA COLLECTION EXPERIMENT

The data collection overhead criteria included goals for measuring session management overhead and data collection overhead. Two independent experiments were conducted to measure the total execution time of the instrumented code. Table 6 describes the parameter settings that were used to support both experiments. The first experiment gathered data using the factors and levels described in Table 7 on a trivial program that performed addition for a specified number of iterations. The number of iterations was chosen such that the program ran for an approximate amount of chosen wall clock time. The experiment used a randomized factorial design and 25 observations for each combination of factors, with a random amount of quiescence (0 to 5 seconds) between runs. The number of events and sampling rate are irrelevant when the sampling mode is off and were ignored when building the randomized run list to ensure an even number of replicates for all configurations. The second experiment collected execution data on real programs. Table 8 lists the factors and levels for collecting data on real programs as opposed to the continuous loop program.

F. AUTOMATED OUTLIER DETECTION

Due to the complex interactions between operating systems and available computer system resources, sequences

TABLE 5. Data collection parameters and settings for evaluation criteria 1, 2, and 3.

Parameters		Settings
Collection Technique	Sample	Minimal Skid Data: IP, TID, Time, CPU, Period Storage Buffer: 9 Pages
Permission Level	User	
Collection Aperture	Process	Pinned to single core, no children
Monitoring Context	Known Code	Known Location
Data Requirements	Learning Mode	N/A
	Data Format	Sequences
Experiment Settings	CPU Frequency	3.2GHz w/performance profile
	Bubblesort	Iterative (No recursion)
	Bubblesort Input	1K Randomly Ordered
	GCC Optimization Level	Default (-O0)
	Fibonacci Implementation	Recursive
	Fibonacci Input	25
	Process Priority	Linux Default
	System Load	Default O/S Services
	Collector Core	Physically Unique from Target

TABLE 6. Parameters and settings to evaluate collection overhead for both sets of data collected for criteria #4.

Parameters		Settings
Collection Technique	Sample	Minimal Skid Data: IP, TID, Time, CPU, Period Storage Buffer: 9 Pages
Chosen Events	Intel	branch-instructions, cache-misses, LLC-stores, ref-cycles
	IBM POWER9	branch-instructions, cache-misses, LLC-stores, L1-icache-loads
Permission Level	User	
Collection Aperture	Process	Pinned to single core, no children
Monitoring Context	Known Code	Known Location
Analytic Data Requirements	Learning Mode	N/A
	Data Format	Sequences
Experiment Settings	CPU Frequency	3.2GHz w/performance profile
	Program	Continuous Loop
	Process Priority	Linux Default
	System Load	Default O/S Services only
	Collector Core	Distinct from Target

collected by any HPC data collection technique will have outliers. The volume of data generated by the collection

TABLE 7. Factors and levels to evaluate collection overhead on simple program for criteria #4.

Factors		Levels
Collection Technique	Poll	
	Sample	
	None	
Measurement Interval	Poll	Time-Based 5 μsec, 10 μsec, 100 μsec
	Sample	Event-Based in Cycles 16K, 32K, 320K Cycles
Experiment Settings	Target Runtime	0 ms, 10 ms, 1 sec, 30 sec

TABLE 8. Factors and levels to evaluate collection overhead for MiBench [51] programs for criteria #4.

Factors		Levels
Collection Technique	Poll	
	Sample	
	None	
Measurement Interval	Poll	Time-Based 5 μsec, 10 μsec, 100 μsec
	Sample	Event-Based in Cycles 16K, 32K, 320K Cycles
Experiment Settings	Program	cjpeg, djpeg, lame, typeset
	Input Size	Small, Large

mechanisms alongside the number of replicates for experiments makes manual outlier detection infeasible. Analyzing the data with statistical tools is helpful to recognize shortcomings in an HPC data collection approach.

This section describes two approaches to manage different kinds of data which might have outliers. The first kind of data are sequences, which record a fixed-length vector of values every timestep. Second, there are individual point values which record properties such as total execution time or mean HPC value for a sequence.

1) COOK's DISTANCE

Linear regression is a technique to describe the relationship of variables using linear parameters. Equation 8 shows a simple linear regression model where X_i holds the ith value of a predictor variable and Y_i records ith observation. β_0 is an intercept term, B_1 is the slope of the regression line (e.g., change in mean value of Y per unit increase in X, and ϵ_i is a random error term [52].

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i \tag{8}$$

For an ideal collector, the elapsed time between measurements (e.g., interarrival time) should be constant. In an ideal collector, when X_i is the index of the ith timestep and Y_i is the

elapsed time then $\beta_1 = 0$. Fitting a linear regression model to the observed data produces an estimate for β_1 that is nonzero.

Cook's Distance is a technique used in linear regression to detect the influence of each X_i observation on the estimate for β_1. When the Cook's Distance value exceeds a threshold of $\frac{4}{n}$, where n is the number of timesteps in the sequence, then the outlier elapsed time is replaced with the sequence's median elapsed time. Cook's distance is only used during the analysis of the elapsed time and therefore the magnitudes of HPCs are not corrected with this approach. The correction for possible event count errors is handled using the second outlier detection approach. The median value was chosen over the mean because of the sharply peaked nature of the distribution.

2) MODIFIED INNER QUARTILE RANGE

The Modified Inner Quartile Range (IQR) technique [53] is used to identify and remove outliers from a population. In this research, the population may be within a single series (e.g., outlier cache misses) or a set of point values describing multiple series. The lower threshold for removal is $Q_1 - 3.0\text{IQR}\left(1 + 0.1 \log \frac{n}{10}\right)$ and the upper threshold is $Q_3 + 3.0\text{IQR}\left(1 + 0.1 \log \frac{n}{10}\right)$, where n is the sample size. The modified IQR technique identifies outliers from the population mass, with consideration for the sample size n.

The outlier detection and remedial measures are specific to the evaluation goals in this research. Applications of the framework to develop deep learning models may choose different outlier detection approaches and remedial measures or may choose to eliminate the requirement for outlier detection.

G. PERFORM EXPERIMENTAL ANALYSIS

The data collection architecture described in Section V was implemented as a general purpose tool designed to run on multiple CPUs with the possibility of extension to larger-scale systems. To demonstrate the usefulness of the framework, the evaluation criteria described in Section VI were applied to three different CPU architectures. The IBM POWER9 and Intel Xeon E5 processors are server-grade CPUs while the Intel i7-4700MQ is intended for use in mobile computers such as laptops. The results of the experimental analysis are presented and discussed in Section VII.

VII. RESULTS

This section presents the results for three different microarchitectures. To provide insight about the fundamental behavior of various approaches to gathering data with HPCs, the results focused on the criteria established in Section VI. These results establish reference data for the application of this research framework so that others can propose improvements to the various collection techniques or system architecture.

A. CRITERIA #1: STATIONARITY OF ELAPSED TIME

In contrast to domains where data is sampled at a fixed rate, the operating system scheduler takes in numerous factors to determine when each task executes. Therefore, the elapsed time is actually a measurement of the underlying control process. Table 9 shows the results of three measurements related to the stationarity of the elapsed time. An important point of consideration when interpreting these results is that the sampling technique and polling technique have distinct frames of reference for determining when to record a measurement. The sampling technique interrupts the CPU after a fixed number of counted cycles whereas the polling technique relies on the operating system scheduler to grant the polling process time on the CPU to read the HPC values.

The value shown for ADF failure in Table 9 captures the ratio of sequences where the presence of a unit root failed to be rejected at $\alpha = 0.05$. Lower ADF ratios indicate a more ideal collection processes. A unit root is an indicator that another property of the computer execution environment influenced the achievement of the desired collection interval. ADF is a low-power test for short sequences (e.g., 100 μsec) but the nonstationarity of sampling technique sequences when compared with the polling technique could be a source of concern. Nevertheless, research that builds systems to use time series HPC data should discuss the stationarity of raw data and corrections for nonstationarity.

Figure 5 shows examples of stationary and nonstationary behavior in traces gathered every 10 μsec (polling) or every 32K clock cycles (sampling). Figure 5b exhibits an undesirable amount of variation and was clearly given a lower execution priority given the high elapsed time at the beginning. The default Linux completely fair scheduler corrects the delayed runtime by granting earlier execution for a period of time. The stair step behavior shown in Figure 5d is also intriguing. The stair stepping likely occurs because the CPU frequency increased at those points in time. The results shown in Figure 5 demonstrate the kinds of shifts statistical models or neural networks must have the capacity to handle.

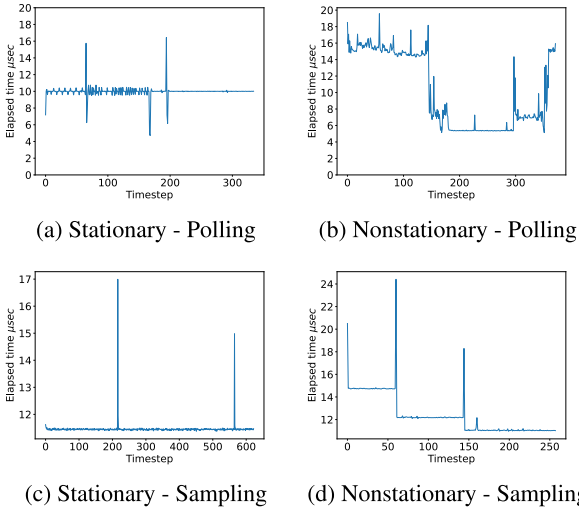
The second important result to draw from Table 9 is the upper-limit for data collection on each of the three systems used. With 95% confidence, the IBM POWER9 average collection interval using the polling technique is between 6.39 μsec and 6.41 μsec when a 5 μsec collection interval is specified. Clearly, the collection interval is not being satisfied, which could lead to increased noise and unexpected results during analysis. In contrast, the Intel systems were reasonably close to the collection interval even at 5 μsec.

B. CRITERIA #2: DATA LOSS

The data loss criteria was evaluated by considering the total number of branch instruction events counted for multiple runs of the same program given the same input. Table 10 shows the 95% confidence interval for the observed standard deviations of total branch instruction events. Recall that an ideal technique should have no variation because the number of branch instructions should be invariant between runs of the chosen program. The sampling technique violates this property with huge standard deviations across all program runs. In contrast, the poll technique experienced fairly small deviations. There was no variation for the IBM POWER9 on

TABLE 9. Stationarity of elapsed time and collection interval statistics on all considered CPUs.

			5 μsec	10 μsec	100 μsec
IBM POWER9	ADF Fail Ratio	Sample			
		fibonacci	0.047	0.131	0.428
		sort	0.007	0.000	0.000
		Poll			
		fibonacci	0.000	0.002	0.376
		sort	0.114	0.000	0.002
	Interval 95% CI	Sample			
		fibonacci	[6.39, 6.41]	[11.42, 11.44]	[101.86, 101.95]
		sort	[6.39, 6.41]	[11.41, 11.43]	[101.79, 101.81]
		Poll			
		fibonacci	[6.74, 6.75]	[9.99, 10.00]	[99.85, 99.94]
		sort	[6.91, 6.93]	[10.00, 10.00]	[99.99, 100.02]
	Std Dev 95% CI	Sample			
		fibonacci	[0.11, 0.14]	[0.20, 0.26]	[0.59, 0.84]
		sort	[0.22, 0.31]	[0.28, 0.30]	[0.89, 0.99]
		Poll			
		fibonacci	[0.26, 0.30]	[0.21, 0.26]	[1.51, 2.26]
		sort	[0.27, 0.29]	[0.18, 0.20]	[2.15, 2.76]
Intel Xeon X5	ADF Fail Ratio	Sample			
		fibonacci	0.01	0.579	-
		sort	0.022	0.035	0.396
		Poll			
		fibonacci	0.045	0.025	0.022
		sort	0	0.007	0.141
	Interval 95% CI	Sample			
		fibonacci	[6.22, 6.23]	[11.42, 11.43]	[104.59, 104.65]
		sort	[6.20, 6.21]	[11.38, 11.39]	[104.54, 104.56]
		Poll			
		fibonacci	[5.54, 5.56]	[9.92, 9.95]	[99.28, 99.47]
		sort	[5.48, 5.49]	[9.95, 9.99]	[99.92, 99.98]
	Std Dev 95% CI	Sample			
		fibonacci	[0.06, 0.09]	[0.06, 0.09]	[0.03, 0.10]
		sort	[0.11, 0.13]	[0.13, 0.16]	[0.09, 0.15]
		Poll			
		fibonacci	[0.32, 0.37]	[0.46, 0.54]	[1.47, 2.20]
		sort	[0.29, 0.31]	[0.28, 0.31]	[2.13, 2.78]
Intel i7-4700MQ	ADF Fail Ratio	Sample			
		fibonacci	0.421	0.403	-
		sort	0.743	0.938	0.614
		Poll			
		fibonacci	0.312	0.027	0.25
		sort	0.52	0.01	0.174
	Interval 95% CI	Sample			
		fibonacci	[7.75, 7.84]	[14.05, 14.24]	[124.24, 126.64]
		sort	[7.29, 7.37]	[12.67, 12.77]	[114.33, 114.89]
		Poll			
		fibonacci	[5.59, 5.73]	[9.94, 9.97]	[98.88, 99.43]
		sort	[5.17, 5.30]	[9.98, 10.03]	[99.33, 99.84]
	Std Dev 95% CI	Sample			
		fibonacci	[0.89, 0.98]	[1.32, 1.48]	[2.04, 2.93]
		sort	[1.09, 1.14]	[1.85, 1.92]	[14.25, 14.69]
		Poll			
		fibonacci	[0.47, 0.54]	[1.06, 1.14]	[2.23, 3.07]
		sort	[0.57, 0.62]	[0.86, 0.93]	[2.69, 5.11]



(a) Stationary - Polling (b) Nonstationary - Polling

(c) Stationary - Sampling (d) Nonstationary - Sampling

FIGURE 5. Examples of stationary and nonstationary elapsed time behavior.

TABLE 10. 95% confidence interval for the std deviation of total event counts for # of branch instructions.

		5 μsec	10 μsec	100 μsec
IBM POWER9	Sample			
	fibonacci	[1018, 1155]	[1493, 2031]	[2124, 2815]
	sort	[99908, 126725]	[733, 889]	[3687, 5145]
	Poll			
	fibonacci	[1.99, 2.66]	[1.62, 2.43]	[0, 0]
	sort	[2.19, 2.84]	[1.88, 2.63]	[0, 1.52]
Intel XEON E5	Sample			
	fibonacci	[0.62, 0.88]	[1042, 1788]	[76152, 79555]
	sort	[397137, 448860]	[211406, 239689]	[30108, 36774]
	Poll			
	fibonacci	[2.78, 3.31]	[0.57, 0.81]	[0.31, 1.40]
	sort	[8.75, 11.93]	[3.90, 5.48]	[0.62, 0.93]
Intel i7-4700MQ	Sample			
	fibonacci	[15.85, 23.84]	[4007, 4732]	[73644, 79608]
	sort	[379987, 452736]	[283798, 311182]	[28055, 32614]
	Poll			
	fibonacci	[8.49, 10.26]	[7.61, 9.74]	[0.90, 1.12]
	sort	[19.53, 26.09]	[12.09, 16.50]	[1.22, 1.85]

the fibonacci program at the 100 μsec collection interval. In machine learning applications, a benefit of minimizing variation in the total event count is that smaller models operating on a vector of features could be used alongside sequence classification approaches.

The reasons causing variation are outside the scope of this analysis. It is plausible that failing to save and restore HPC state from the kernel during context switch could include branch instructions from other programs [20]. Reflecting on Basu's hypothesis that CFGs could be confirmed using HPC data, the sampling technique seems to present significant challenges. However, the polling technique presents a less obvious reason for concern. Spectre and Meltdown attack detection research could be impacted by the large variation

TABLE 11. 95% confidence interval for distribution-free Kolmogorov-Smirnov test between any two sequences.

		5 μsec	10 μsec	100 μsec
IBM POWER9	Sample			
	fibonacci	[0.940, 0.945]	[0.954, 0.958]	[0.970, 0.976]
	sort	[0.727, 0.735]	[0.818, 0.826]	[0.988, 0.989]
	Poll			
	fibonacci	[0.823, 0.834]	[0.939, 0.948]	[0.970, 0.976]
	sort	[0.869, 0.873]	[0.814, 0.820]	[0.946, 0.955]
Intel XEON E5	Sample			
	fibonacci	[0.794, 0.808]	[0.944, 0.952]	-
	sort	[0.821, 0.832]	[0.906, 0.914]	[0.973, 0.977]
	Poll			
	fibonacci	[0.490, 0.510]	[0.495, 0.518]	[1.000, 1.000]
	sort	[0.686, 0.697]	[0.820, 0.835]	[0.958, 0.964]
Intel i7-4700MQ	Sample			
	fibonacci	[0.803, 0.818]	[0.886, 0.898]	-
	sort	[0.745, 0.760]	[0.792, 0.806]	[0.957, 0.962]
	Poll			
	fibonacci	[0.155, 0.168]	[0.317, 0.339]	[0.854, 0.873]
	sort	[0.457, 0.473]	[0.404, 0.417]	[0.848, 0.857]

TABLE 12. 95% confidence interval for dynamic time warping measurement between two sequences.

		5 μsec	10 μsec	100 μsec
IBM POWER9	Sample			
	fibonacci	[1018, 1113]	[866, 871]	[2261, 2494]
	sort	[1315, 1363]	[1026, 1115]	[1691, 1701]
	Poll			
	fibonacci	[1725, 1748]	[1069, 1141]	[3457, 4214]
	sort	[1762, 1769]	[1621, 1638]	[9002, 9366]
Intel XEON E5	Sample			
	fibonacci	[336, 338]	[318, 320]	[1174, 1312]
	sort	[1971, 2039]	[2633, 2687]	[5994, 6245]
	Poll			
	fibonacci	[780, 808]	[450, 457]	[11047, 13346]
	sort	[2236, 2263]	[3514, 3615]	[20744, 21402]
Intel i7-4700MQ	Sample			
	fibonacci	[1223, 1521]	[991, 1290]	[18781, 20719]
	sort	[2298, 2457]	[3723, 3890]	[8971, 9269]
	Poll			
	fibonacci	[4305, 4511]	[13079, 13610]	[63009, 65525]
	sort	[3332, 3717]	[7722, 7848]	[39498, 40709]

if the sampling technique is used because the variation could make it easier for low-rate attacks to evade detection.

C. CRITERIA #3: SEQUENCE VARIATIONS

Two different techniques were used to measure the sequence variation between runs. Table 11 presents the confidence intervals for the mean ratio of failures to reject H_0 (at least one point comes from a different distribution) to the number of windows compared in the sequence. A ratio near 1.0 in Table 11 indicates the collection technique yields similar distributions of event counts across all windows in the sequence.

In general, the sampling technique produces more similar distributions. Given the data loss issues described in Section VII-B, this similarity could be a result of events not being counted. The data for 100 μsec on the Intel processors were omitted because the sequences contained too few samples. The similarity of the polling sequences at 5 and 10 μsec is fairly low, especially for the CPU-bound fibonacci task. Unfortunately, this result suggests that faster collection rates (which imply stronger relationship to the CFG in [17]) might experience more than 5% nondeterminism when analyzed as a sequence.

The second approach to assess the similarity of two sequences calculated 95% confidence intervals for the DTW measurement between any two randomly chosen sequences. The results of the DTW calculation are provided in Table 12. There are 4 observations to be made. First, the smallest collection interval achievable on the processor architecture tends (with some exceptions) to produce the smallest DTW distance between two sequences for the polling and sampling techniques. On the Intel processors, this suggests that 5 μsec is a reasonable collection interval while 10 μsec intervals are reasonable on the IBM POWER9.

TABLE 13. Observed overhead after subtracting the mean management overhead for the collection technique.

	Poll (μsec)		Sample (μsec)	
	Mean	Range	Mean	Range
Management	16.95	[1, 35]	1018.55	[1002, 1039]
10 ms Duration				
5μsec	2.85	[-14, 17]	168.85	[-29, 1570]
10μsec	3.66	[-15, 54]	-13.22	[-29, 13]
100μsec	-0.21	[-16, 17]	-11.35	[-29, 6]
1 sec Duration				
5μsec	302.65	[258, 397]	174.12	[136, 210]
10μsec	192.90	[155, 232]	137.45	[109, 164]
100μsec	17.72	[-5, 74]	-266.42	[-312, -224]
30 ms Duration				
5μsec	5724.83	[4987, 8632]	6059.14	[5137, 10133]
10μsec	5589.72	[4922, 7531]	5214.70	[3693, 8587]
100μsec	484.53	[317, 796]	221.48	[94, 411]

Second, the Intel i7-4700MQ processor (which is designed for laptops) produced noticeably higher DTW measurements for both collection techniques, but the DTW measurement magnitude for polling at 100 μsec is quite large. Analyzing the underlying reasons is beyond the scope of this analysis but could be attributable to mobile-specific power management design trade-offs or different PMU implementations.

Third, the DTW measurement for the CPU-bound fibonacci task tends to be lower than the sort task. Initially, it appears that stochasticity associated with memory accesses could be a source of variation between CPU-bound tasks and those that depend on other resources such as main memory.

TABLE 14. Total polling overhead on real programs.

Interval	Input Size	Program	Mean Slowdown	Slowdown Range
5 μsec	Small	cjpeg	4.27	[1.40, 7.59]
		djpeg	16.66	[1.84, 30.05]
		lame	1.28	[1.17, 1.36]
		typeset	2.06	[1.27, 2.88]
	Large	cjpeg	2.09	[1.22, 3.07]
		djpeg	5.34	[1.38, 8.46]
		lame	1.19	[1.17, 1.21]
		typeset	1.35	[1.22, 1.46]
10 μsec	Small	cjpeg	4.10	[1.33, 7.35]
		djpeg	14.33	[1.88, 28.85]
		lame	1.27	[1.17, 1.40]
		typeset	1.91	[1.26, 2.80]
	Large	cjpeg	2.01	[1.24, 2.47]
		djpeg	5.14	[1.37, 8.46]
		lame	1.19	[1.17, 1.22]
		typeset	1.36	[1.22, 1.52]
100 μsec	Small	cjpeg	4.04	[1.33, 6.76]
		djpeg	14.32	[1.86, 26.46]
		lame	1.28	[1.17, 1.36]
		typeset	1.94	[1.24, 2.81]
	Large	cjpeg	2.10	[1.22, 2.89]
		djpeg	4.89	[1.37, 8.47]
		lame	1.19	[1.17, 1.21]
		typeset	1.35	[1.22, 1.48]

TABLE 15. Total sampling overhead on real programs.

Interval	Input Size	Program	Mean Slowdown	Slowdown Range
5 μsec	Small	cjpeg	196.55	[192.96, 199.97]
		djpeg	878.50	[854.29, 892.77]
		lame	7.29	[7.11, 7.55]
		typeset	51.40	[50.38, 52.35]
	Large	cjpeg	60.06	[59.03, 60.99]
		djpeg	253.02	[247.37, 260.69]
		lame	1.72	[1.69, 1.75]
		typeset	7.31	[1.19, 10.66]
10 μsec	Small	cjpeg	196.27	[192.89, 198.68]
		djpeg	879.04	[854.23, 895.25]
		lame	7.27	[7.10, 7.38]
		typeset	51.41	[50.38, 52.41]
	Large	cjpeg	60.01	[59.03, 61.11]
		djpeg	252.56	[247.46, 255.77]
		lame	1.72	[1.69, 1.75]
		typeset	8.02	[1.30, 10.58]
100 μsec	Small	cjpeg	196.26	[192.90, 199.24]
		djpeg	877.74	[854.22, 894.40]
		lame	7.27	[7.10, 7.36]
		typeset	51.28	[50.36, 52.34]
	Large	cjpeg	59.91	[59.03, 60.54]
		djpeg	252.11	[247.34, 257.94]
		lame	1.72	[1.69, 1.74]
		typeset	6.61	[1.31, 10.60]

However, a direct comparison between fibonacci and sort is unreasonable since the duration each program executes is different and the code executed between branch instructions will lead to different event magnitudes.

Finally, there appears to be disagreement with the KS test results provided in Table 11 and the DTW results provided in Table 12. For example, the polling technique produces low ratios (approx. 50%) but the DTW metric is small. The windowed approach to the KS test draws hard interval boundaries assuming a constant rate progression through the program whereas DTW includes a warping parameter that allows for alignment between signals of different rates. The complexities of executing code make the assumption of the KS test approach unrealistic, while the flexibility of DTW overcomes this limitation.

D. CRITERIA #4: COLLECTION OVERHEAD

This section describes the results of the collection overhead analysis. The results were obtained for all three processors described in this research, but the results were similar enough to present only the IBM POWER9 results. Table 13 shows both the estimated overhead for the management of a data collection session and the time penalty associated with storing data after the estimated time penalty for session management has been removed. The first row in Table 13 shows the estimated session establishment overhead for the polling versus sampling collection techniques. Notably, the sampling technique takes an order of magnitude longer than the polling technique to configure the HPCs. If a system uses a system-wide aperture or monitors long running programs where the configuration of the HPCs happens infrequently, this long delay is likely not a significant factor. However, if the collection aperture narrows to specific threads with known code locations, this configuration penalty could render the approach unusable for practical purposes.

The values reported for mean and range on each of the different durations were calculated by subtracting the estimated average management overhead from the total program execution time. Negative values occur when programs in the treatment group finish faster than the program in the control group. The negative values in Table 13 should be interpreted as indications that the session management and collection process imposes negligible overhead in the average case. For short periods of collection such as a 10 ms program, the overhead of collecting and storing data is trivial. However, the length of data collection increases the amount of time required to save and store data. The data storage task could be significantly improved by proactively writing data to disk at intervals during data collection.

Table 14 shows the total overhead (i.e., the estimated mean configuration time is not subtracted from program execution time) for real programs when the polling technique is used and Table 15 shows the total overhead when the sampling technique is used. For both collection techniques, the average slowdown tends to decrease for programs that take longer to finish executing (e.g., lame and typeset), whereas programs that execute more quickly exhibit greater slowdown because of the management overhead. If session management costs are ignored, then the overhead of data collection is negligible.

VIII. CONCLUSION AND FUTURE WORK

This research proposed a framework to organize numerous settings required to configure HPCs. The framework described is not dependent upon a specific microarchitecture and allows for the flexibility of choosing different implementation tools, while providing greater clarity about precisely how HPCs were configured. The framework informed a data

collection architecture that was implemented to establish a baseline for characterizing the behavior of HPC data as a sequence on real systems. Four evaluation criteria were evaluated in an experimental environment on three different types of CPUs and discussion of the results provides insight for future HPC research to consider.

There are several opportunities for future work, especially when the diversity of use cases for HPC data is considered. First, Basu's hypothesis [17] suggests strong guarantees with HPC data. A key limitation of the approach taken by Basu is the dependence upon a ''golden'' HPC reading. An approach based on empirical data that used variation between sequences as the foundation for analysis while achieving similar results could be of great benefit. Second, the quantity of possible events and relationships to different behaviors of interest are not well described. A methodical approach to run experiments, collect data and make informed decisions about which events to select, would fill an important gap in this research area. Third, in the context of information security use cases, all current HPC techniques assume the operating system is not compromised and part of the trusted computing base. Dedicated hardware to configure PMUs and collect data could provide stronger security properties.

ACKNOWLEDGMENT

The views expressed in this document are those of the authors and do not reflect the official policy or position of the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2021-1021.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, and D. Genkin, ''Spectre attacks: Exploiting speculative execution,'' in *Proc. Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, 2019, pp. 1–19.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, ''Meltdown: Reading kernel memory from user space,'' in *Proc. 27th USENIX Secur. Symp. Secur. (USENIX)*, Baltimore, MD, USA, Aug. 2018, pp. 973–990.

[3] J. Szefer, ''Survey of microarchitectural side and covert channels, attacks, and defenses,'' *J. Hardw. Syst. Secur.*, vol. 3, no. 3, pp. 219–234, Sep. 2019.

[4] C. Canella, K. N. Khasawneh, and D. Gruss, ''The evolution of transient-execution attacks,'' in *Proc. Great Lakes Symp. VLSI*, Virtual Event, China, Sep. 2020, pp. 163–168.

[5] X. Jin and N. Yu, ''A defense mechanism against transient execution attacks on SMT processors,'' *IEICE Electron. Exp.*, vol. 18, p. 10041, Nov. 2021.

[6] C. Reinbrecht, S. Hamdioui, M. Taouil, B. Niazmand, T. Ghasempouri, J. Raik, and J. Sepulveda, ''LiD-CAT: A lightweight detector for cache attacks,'' in *Proc. Eur. Test Symp.*, Tallinn, Estonia, May 2020, pp. 1–6.

[7] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanoviá, and D. Song, ''Keystone: An open framework for architecting trusted execution environments,'' in *Proc. 15th Eur. Conf. Comput. Syst.*, Heraklion, Greece, Apr. 2020, pp. 1–16.

[8] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, ''CURE: A security architecture with customizable and resilient enclaves,'' in *Proc. 30th USENIX Secur. Symp.*, Aug. 2021, pp. 1073–1090.

[9] C. Li and J.-L. Gaudiot, ''Online detection of spectre attacks using microarchitectural traces from performance counters,'' in *Proc. 30th Int. Symp. Comput. Archit. High Perform. Comput.*, Lyon, France, Sep. 2018, pp. 25–28.

[10] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, ''WHISPER: A tool for run-time detection of side-channel attacks,'' *IEEE Access*, vol. 8, pp. 83871–83900, 2020.

[11] B. Zheng, J. Gu, J. Wang, and C. Weng, ''CBA-detector: A self-feedback detector against cache-based attacks,'' *IEEE Trans. Dependable Secure Comput.*, early access, Jun. 16, 2021, doi: 10.1109/TDSC.2021.3089882.

[12] W. Wang, G. Chen, Y. Cheng, Y. Zhang, and Z. Lin, ''Specularizer: Detecting speculative execution attacks via performance tracing,'' in *Detection Intrusions and Malware, and Vulnerability Assessment*, vol. 12756, L. Bilge, L. Cavallaro, G. Pellegrino, and N. Neves, Eds. Cham, Switzerland: Springer, 2021, pp. 151–172.

[13] P. Cronin and C. Yang, ''Lowering the barrier to online malware detection through low frequency sampling of HPCs,'' in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Apr. 2018, pp. 177–180.

[14] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, ''Hardware performance counters can detect malware: Myth or fact?'' in *Proc. Asia Conf. Comput. Commun. Secur.*, Incheon, South Korea, May 2018, pp. 457–468.

[15] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, ''A cautionary tale about detecting malware using hardware performance counters and machine learning,'' *IEEE Design Test*, vol. 38, no. 3, pp. 39–50, Mar. 2021.

[16] C. Malone, M. Zahran, and R. Karri, ''Are hardware performance counters a cost effective way for integrity checking of programs,'' in *Proc. 6th ACM Workshop Scalable Trusted Comput.*, 2011, pp. 71–76.

[17] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, ''A theoretical study of hardware performance counters-based malware detection,'' *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 512–525, 2020.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann, 2017.

[19] M. N. Islam and S. Kundu, ''PMU-Trojan: On exploiting power management side channel for information leakage,'' in *Proc. 23rd Asia South Pacific Design Automat. Conf.*, Jan. 2018, pp. 709–714.

[20] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, ''SoK: The challenges, pitfalls, and perils of using hardware performance counters for security,'' in *Proc. IEEE Symp.*, Oct. 2019, pp. 20–38.

[21] A. Biswas, Z. Li, and A. Tyagi, ''Performance counters and DWT enabled control flow integrity,'' *Social Netw. Comput. Sci.*, vol. 3, no. 1, p. 48, Jan. 2022.

[22] V. Weaver. (2021). *GitHub*. Accessed: Sep. 7, 2021. [Online]. Available: https://github.com/deater/perf_event_tests

[23] O. Aciicmez, ''Yet another microarchitectural attack: Exploiting I-cache,'' Samsung Inf. Syst. Amer., Samsung Electron., San Jose, CA, USA, Tech. Rep. 164, 2007.

[24] C. Percival, ''Cache missing for fun and profit,'' in *Free BSD Presentations Papers*. Ottawa, ON, Canada: BSDCan, 2005. Accessed: Dec. 12, 2021. [Online]. Available: https://papers.freebsd.org/2005/cperciva-cache_missing/

[25] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, ''Last-level cache side-channel attacks are practical,'' in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.

[26] M. Payer, ''HexPADS: A platform to detect 'stealth' attacks,'' in *Engineering Secure Software and Systems* (Lecture Notes in Computer Science), vol. 9639, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Cham, Switzerland: Springer, 2016, doi: 10.1007/978-3-319-30806-7_9.

[27] V. M. Weaver and S. A. McKee, ''Can hardware performance counters be trusted?'' in *Proc. Int. Symp. Workload Characterization*, 2008, pp. 141–150.

[28] V. M. Weaver, D. Terpstra, and S. Moore, ''Non-determinism and overcount on modern hardware performance counter implementations,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2013, pp. 215–224.

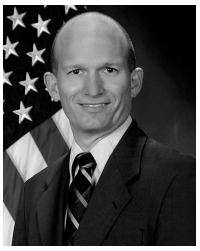
[29] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, ''On the feasibility of online malware detection with performance counters,'' *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, 2013.

[30] M. Kazdagli, V. J. Reddi, and M. Tiwari, ''Quantifying and improving the efficiency of hardware-based mobile malware detectors,'' in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.

[31] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, ''On the detection of kernel-level rootkits using hardware performance counters,'' in *Proc. Asia Conf. Comput. Commun. Secur.*, Abu Dhabi, United Arab Emirates, Apr. 2017, pp. 483–493.

[32] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research Attacks, Intrusions and Defenses*, vol. 8688, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Cham, Switzerland: Springer, 2014, pp. 109–129.
[33] N. Herath and A. Fogh. (2015). Not Your Grand Daddy's CPU Performance Counters—CPU Hardware Performance Counters for Security. Black Hat 2015 Briefings. Accessed: Nov. 20, 2020. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf
[34] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 27–38.
[35] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Proc. 41st Int. Symp. Comput. Archit.*, Minneapolis, MN, USA, Jun. 2014, pp. 361–372.
[36] S. Das, B. Chen, M. Chandramohan, Y. Liu, and W. Zhang, "ROPSEntry: Runtime defense against ROP attacks using hardware performance counters," *Comput. Secur.*, vol. 73, pp. 374–388, Oct. 2018.
[37] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (Lecture Notes in Computer Science), vol. 9721, J. Caballero, U. Zurutuza, and R. Rodríguez, Eds. Cham, Switzerland: Springer, 2016, doi: 10.1007/978-3-319-40667-1_14.
[38] M. Alam, S. Bhattacharya, and D. Mukhopadhyay, "Victims can be saviors: A machine learning–based detection for micro-architectural side-channel attacks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 2, pp. 1–31, Jan. 2021.
[39] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in," in *Proc. 23rd USENIX Symp.*, 2014, p. 20.
[40] Y. Xia, J. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proc. Int. Conf. Dependable Syst. Netw.*, Boston, MA, USA, Jun. 2012, pp. 1–12.
[41] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. Accessed: Dec. 12, 2021. [Online]. Available: http://www.deeplearningbook.org
[42] (2018). *POWER9 Performming Monitor Unit User's Guide, V1.2*. Accessed: Oct. 20, 2020. [Online]. Available: https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf
[43] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Research in Attacks, Intrusions, and Defenses* (Lecture Notes in Computer Science), vol. 9854, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Cham, Switzerland: Springer, 2016, doi: 10.1007/978-3-319-45719-2_6.
[44] C. Ashford. (2021). *LWN*. Accessed: Nov. 24, 2021. [Online]. Available: https://lwn.net/Articles/370414/
[45] G. James, R. T. Daniela Witten, and T. Hastie, *An Introduction to Statistical Learning: With Applications in R*. New York, NY, USA: Springer, 2017.
[46] S. Chadha, D. A. Hrusecky, and D. Q. Nguyen, "Parallel slice processor having a recirculating load-store queue for fast deallocation of issue queue entries," U.S. Patent 10 133 576, Nov. 20, 2016.
[47] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting Control*. Hoboken, NJ, USA: Wiley, 2015.
[48] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric Statistical Methods*, 3rd ed. Hoboken, NJ, USA: Wiley, 2014.
[49] R. Tavenard, J. Faouzi, G. Vandewiele, F. Divo, G. Androz, C. Holtz, M. Payne, R. Yurchak, M. Rußwurm, K. Kolar, and E. Woods, "Tslearn, a machine learning toolkit for time series data," *J. Mach. Learn. Res.*, vol. 21, no. 118, pp. 1–6, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-091.html
[50] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-26, no. 1, pp. 43–49, Feb. 1978.
[51] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. Int. Workshop Workload Characterization*, Austin, TX, USA, 2001, pp. 3–14.
[52] M. H. Kutner, *Applied Linear Statistical Models*. Boston, MA, USA: McGraw-Hill, 2005.
[53] G. Barbato, E. M. Barini, G. Genta, and R. Levi, "Features and performance of some outlier detection methods," *J. Appl. Statist.*, vol. 38, no. 10, pp. 2133–2149, Oct. 2011.

TOR J. LANGEHAUG (Student Member, IEEE) received the B.S. degree in computer science from California State University, Sacramento, CA, USA, in 2009, and the M.S. degree in information security from Carnegie Mellon University, Pittsburgh, PA, USA, in 2017. He is currently pursuing the Ph.D. degree with the Air Force Institute of Technology, investigating applications of machine learning to low-level microarchitectural data sources.



SCOTT R. GRAHAM (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2004. He is currently an Associate Professor of computer engineering with the Air Force Institute of Technology, Wright-Patterson Air Force Base, Dayton, OH, USA. His research interests include the security of cyber–physical systems, computer architecture, networks, and security.

CHRISTINE M. SCHUBERT KABBAN received the B.S. degree in mathematics from the University of Dayton, Dayton, OH, USA, in 1992, the M.S. degree in applied statistics from Wright State University, Dayton, in 1995, and the Ph.D. degree in applied mathematics from the Air Force Institute of Technology (AFIT), Wright-Patterson Air Force Base, OH, USA, in 2005.

She was an Assistant Professor with Virginia Commonwealth University, Richmond, VA, USA. She is currently a Professor of statistics with AFIT. She has been researching and practicing statistics for over 20 years in clinical, engineering, and statistical fields. Her research interests include applications to structural health monitoring, target detection, and autonomous systems and networks with hierarchical and complex multidimensional data.

BRETT J. BORGHETTI earned a Ph.D. degree in computer science in 2008 from the University of Minnesota, Twin Cities, MN; a M.S. degree in computer systems in 1996 from the Air Force Institute of Technology (AFIT) in Dayton, OH; and a B.S. degree in electrical engineering in 1992 from the Worcester Polytechnic Institute (WPI), Worcester, MA.

He is an Associate Professor in the Department of Electrical and Computer Engineering in the Graduate School of Engineering Management at the Air Force Institute of Technology. His research interests focus on improving human–machine team performance and exploiting sensed information in complex environments using artificial intelligence and machine learning. He has research experience in estimating human cognitive performance, statistical machine learning, genetic algorithms, self-organizing systems, neural networks, game theory, information theory and cognitive science.