Classifying Co-resident Computer Programs Using Information Revealed by Resource Contention

TOR LANGEHAUG*, BRETT BORGHETTI*, and SCOTT GRAHAM*, Air Force Institute of Technology

Modern computer architectures are complex, containing numerous components that can unintentionally reveal system operating properties. Defensive security professionals seek to minimize this kind of exposure while adversaries can leverage the data to attain an advantage. This article presents a novel covert interrogator program technique using light-weight sensor programs to target integer, floating point, and memory units within a computer’s architecture to collect data that can be used to match a running program to a known set of programs with up to 100% accuracy under simultaneous multithreading conditions. This technique is applicable to a broad spectrum of architectural components, does not rely on specific vulnerabilities, nor requires elevated privileges. Furthermore, this research demonstrates the technique in a system with operating system containers intended to provide isolation guarantees that limit a user’s ability to observe the activity of other users. In essence, this research exploits observable noise that is present whenever a program executes on a modern computer. This article presents interrogator program design considerations, a machine learning approach to identify models with high classification accuracy, and measures the effectiveness of the approach under a variety of program execution scenarios.

CCS Concepts: • Computer systems organization → Architectures; • Security and privacy → Side-channel analysis and countermeasures; • Computing methodologies → Supervised learning by classification;

Additional Key Words and Phrases: Computer security, side channels, covert channels, machine learning

ACM Reference format:
Tor Langehaug, Brett Borghetti, and Scott Graham. 2023. Classifying Co-resident Computer Programs Using Information Revealed by Resource Contention. Digit. Threat.: Res. Pract. 4, 2, Article 17 (August 2023), 29 pages.
https://doi.org/10.1145/3464306

1 INTRODUCTION

Modern computer architectures are sophisticated with many performance optimizations that an adversary can use to unintentionally reveal information about computer system state, creating a rich environment for machine learning applications. Sources such as network timing data, disk or memory read/write times, temperature sensors, and many others implicitly carry information about a computer system’s operational state. Defensive security experts are concerned about information revealed through these channels while adversaries seek to exploit the channels with discreet and undetectable means. For example, a security architect for a cloud service provider wishes to design an architecture where untrusted entities share resources safely without revealing information about any other user. An adversary targeting critical infrastructure might be interested in identifying

*Disclaimer: The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2020-2674.
Authors’ address: T. Langehaug, B. Borghetti, and S. Graham, Air Force Institute of Technology, AFIT/ENG, 2950 Hobson Way, WPAFB, Ohio, 45433; emails: {tor.langehaug, brett.borghetti, scott.graham}@afit.edu.
This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.
2023. 2576-5337/2023/08-ART17 $15.00
https://doi.org/10.1145/3464306
vulnerabilities or sensing a mission critical workload beginning so proactive measures can be taken autonomously without explicit command and control.

This research focuses on a potential adversarial use case, where a covert observer wishes to identify a specific program running in a shared computing environment by capturing a fingerprint of the program’s influence on another program under the observer’s control and matching that fingerprint to one in a set of previously collected fingerprints of many programs’ behavior. This research uses a controlled execution environment to describe a straightforward interrogator program design that uses count measurements of completed work to classify a simultaneously running program under a variety of code execution scenarios. The collected data are used to develop machine learning models that can accurately match new collections of a program’s fingerprint to known fingerprints. This process assumes the attacker is able to run code on the target system inside an operating system container but does not require elevated permissions, access to restricted application programming interfaces (API), nor access to system information available through the file system such as /procfs [23] and /sysfs [24].

The interrogator program and machine learning models presented in this article are capable of matching data to one of 10 known fingerprints with up to 100% accuracy when code runs in the simultaneous multithreading (SMT) context with only the interrogator and victim program running side-by-side using a five-second observation period. Additionally, this research explores several factors that contribute to the effectiveness of the proposed approach. Several works have been published applying machine learning to perform inference attacks in handheld mobile computing and are discussed in Section 3. It is not uncommon for application inference classification models to report near 100% accuracy on data collected passively by monitoring an information channel [6, 34, 38]. This research is unique in the application of an active approach to collect information where the adversary is not reliant on resources or system calls outside her control. Rather, the adversary executes code to create conditions for information gathering in a shared computing environment.

This research focuses on statistical machine learning techniques rather than applying deep neural networks. The statistical machine learning approaches used in this article are described in Reference [18]. The described approaches have robust statistical underpinnings that lead to enhanced interpretation, require fewer computational resources to train/evaluate, and tend to be effective with less data. In contrast, neural networks often rely on complex network architectures, sophisticated hyperparameter tuning, and significantly more data [11]. Given the exploratory nature of this research, the statistical machine learning approaches provide a rich set of tools to explore many different kinds of models. This approach may not always lead to the highest-performing classifiers that generalize well, but allows consideration of more system execution properties and how those properties could leak information about running programs.

1.1 Problem Background and Motivation

To motivate the research problem, this section describes a simple interrogation scheme that an observer (Eve) could use to obtain private information of another user (Alice). The interrogator program used in this research extends this basic scheme. In Figure 1, Alice and Eve share computing resources using operating system containers that prevent users from having knowledge about the activities of other users on the system. Alice uses a simple program to process her sensitive data by performing several computations when her private data contains the bit value “1,” but executes several no-operations when her private data contains a “0.” Eve wishes to learn Alice’s private information so Eve implements a program running expensive operations continuously. The computer architecture interleaves thread execution, giving the appearance of concurrency, so Alice’s program execution introduces observable changes in Eve’s interrogator behavior, such as the number of operations completed in a fixed period of time. Therefore, Eve can observe the reduced performance in her program’s execution when Alice has the value “1” in her private data, revealing information across a security boundary.

Every program that runs on a modern computer architecture inherently impacts the system in ways that can be sensed by another program running on the same hardware, potentially violating security policy. In this research,
Eve is the adversarial observer, implementing an active interrogator program to determine the application Alice is running. The approach presented in this article observes information from three execution units as sensors and extracts eleven features per sensor from collected data to classify 10 different programs.

1.2 Research Questions
Using the general information leakage scheme described in Section 1.1 and Figure 1, this research explores the following research questions:

1. Using six statistical machine learning models (described in Section 4.4), which model will most accurately match unseen data collected by the interrogator program to a set of known fingerprints?
2. Under what execution conditions can runtime observations lead to greater than 90% accuracy, and which features provide the most useful information to identify a program?

1.3 Research Hypotheses
For each research question, a hypothesis was developed. The insights gained from model development and performance measurement are used to evaluate these hypotheses in Section 6.

1. The classifier that performs best on these data will adapt to non-normality in the dataset. Interrogator design experiments revealed different data distributions by program classes that were bimodal and skewed.
2. The execution conditions where a program has the greatest degree of interaction with another process will yield the highest classification accuracy results. In other words, code that is interleaved with other code will result in higher classification accuracy than code running on different cores and sharing only the memory system.

1.4 Summary of Contributions
This research makes three contributions in the area of applying machine learning to information security:

1.4.1 Demonstration of an Active Interrogator Design. This research demonstrates how an active observer with minimal privileges can interrogate a program using various sensors to exploit the computer system’s shared
architecture. The observer collects an initial set of data to develop a machine learning model that can accurately recognize one of 10 programs the model is designed to identify. The observer then evaluates how accurately the model performs when new data, that were not used to develop the model, is provided as input. The model development is approached as a machine learning classification problem using statistical machine learning techniques, operating with relatively small amounts of data and significantly outperforming random chance guessing. This active approach is in contrast to passively collecting data using operating system services that are easily restricted by requiring elevated privileges. The interrogator controls the volume and rate of data collection.

1.4.2 Use of Machine Learning to Understand Security Implications of Shared Execution Environments. The research presented in this article explores the system operating characteristics that could lead to a decrease in classification accuracy using the interrogator program’s data. System designers, thinking through the lens of an adversary, can use this information to make informed decisions about workload balancing and partitioning resources.

1.4.3 Description of Several Layers of the Computer System that Contribute to Information Leakage. This research takes a holistic view of how user code executes in hardware. This view includes writing the software in a high-level language, compiling behavior of the operating system facilities to schedule code, and processor-specific details designed to optimize code performance. The summary of computer system organization presented in Section 2 facilitates understanding why the technique proposed in this article works. The remainder of the document is structured as follows: Section 2 provides a detailed discussion of computer system organization that make the design of an interrogator program possible. Section 3 discusses related research on side channels, covert channels, and application inference using machine learning. Section 4 describes the methodology for data collection, feature extraction, model selection and performance evaluation. Section 5 presents the results and discusses the execution environments where machine learning techniques lead to high-accuracy results, and where the classification task is less accurate. Section 6 concludes the article and considers opportunities for future work.

2 COMPUTER SYSTEM ORGANIZATION

The overall architecture for modern computing systems is complex. This section introduces basic concepts related to multicore microprocessors, approaches to simultaneous instruction execution, and how operating systems assign code to run on the Central Processing Unit (CPU). Computer architecture is vast area of study. This background section is not intended to exhaustively explore the domain but to offer a sufficient level of knowledge to broadly understand why the techniques discussed in this article are possible.

2.1 Microprocessor Architecture

Modern microprocessors are designed to optimize performance by leveraging many techniques simultaneously to maximize instruction throughput while satisfying thermal design constraints [15]. This optimization results in unintended security consequences that are difficult to eliminate without substantially reducing system performance. Information security professionals and system designers who understand the risks posed by the underlying architecture can make decisions that satisfy security design goals.

Figure 2 introduces the Intel Core i7 processor family architecture that is commonly used in desktops and mobile platforms. Although the Core i7 is in its 11th generation, the basic architecture shown in Figure 2 is common across generations. The changes that occur between generations typically include changes to the number of transistors, register widths, cache size, operating frequency, or size of the reorder buffer [7].

Figure 2 shows a four-physical core processor with eight logical cores. In the Linux operating system, the four physical cores are presented to system applications and users as eight distinct cores [4] even though each pair
Fig. 2. Intel i7 high-level architecture [7].

Fig. 3. Common approaches to parallel execution [15].

Table 1. Summary of Shared Units

| Sub-Unit                | Shared Unit                        |
|-------------------------|------------------------------------|
| Logical Core            | L1 and L2 Cache                    |
| Logical Core            | Execution Engine                   |
| Physical Core L3 Cache  | L3 Cache                           |
| Physical Core           | QPI Interface, Integrated Memory Controller |

of logical cores is supported by a single Level 1 (L1) cache, Level 2 (L2) cache, and execution engine. Table 1 summarizes the high-level locations shown in Figure 2 where resource sharing occurs.

2.2 Parallel Instruction Execution

Parallel instruction execution is a technique modern microprocessors employ to increase instruction throughput. Hennessy and Patterson summarize the four common approaches as superscalar, coarse-grained multithreading, fine-grained multithreading, and SMT [15]. Figure 3 provides a visual representation of the differences in approach, where the shading represents different threads.

Superscalar architectures achieve instruction level parallelism by issuing multiple instructions in a single clock cycle. In the simple superscalar approach, the executing instructions come from only one thread. Stalls in the superscalar pipeline result in periods of time where the processor is unused. The three subsequent multithreading approaches build upon the superscalar multiple-issue idea while trying to reduce unused clock cycles. The coarse-grained multi-threading switches between running threads only when a stall incurs a significant execution delay.
A fine-grained multithreading approach switches between threads on every clock cycle. SMT is a natural outcome of fine-grained multithreading, dynamic scheduling, and register renaming [15]. The SMT approach is common to most modern high-end processors and Intel uses the term Hyper-Threading to describe SMT [7, 15].

A key insight from Figure 2 is that the logical processor concept creates a convenient abstraction for operating systems to leverage SMT. Intel and Advanced Micro Devices processors allow for one or two logical processors (threads) per core [2, 7] while IBM’s POWER9 offers one, two, or four logical processors per core [17].

Logical processors, using shared physical resources, create resource bottlenecks that can lead to unintentional information leakage. The resource bottlenecks that exist at the microarchitectural level make it possible for one piece of code to observe the side effects, typically through a timing channel, of executing multi-threaded code. Additional performance optimizations such as pre-fetching, speculative execution, and branch prediction add complexity to the ways in which code interacts with other code in physical hardware, without the programmer’s knowledge.

Figure 4 presents a slightly simplified view of a single core on the Intel i7-6700 processor [15] to illustrate the complexity of the interactions. Although Figure 4 separates the pieces into four stages, the actions within each stage may take several cycles to complete. Some potential bottlenecks that are apparent in Figure 4 are associated with number of execution units to support the Core i7’s six instruction issue width. There are a fixed number of arithmetic logic unit (ALU), load, store, floating point, and single instruction multiple data (SIMD) execution units that could lead to information leakage when threads are scheduled to different logical cores but share the same physical execution units. The resource contentions also exists in the on-core memory system (L1 and L2 cache) and the off-core memory system (Level 3 cache). An exhaustive detailing of all possible points of interaction is outside the scope of this research. However, recent transient execution attack research [5] provides a rich foundation for understanding the variety of ways in which microarchitectural features are able to be exploited.

2.3 Operating System Scheduler

The operating system’s scheduler plays an important role in reducing or enabling information leakage. The operating system scheduler determines which processors execute a particular task, how long a task is allowed to run, and in the case of SMT, which tasks are allowed to run concurrently on shared hardware. These risks are particularly acute in the case of SMT where L1 and L2 caches are shared by two or more different tasks. To minimize cache-based information leakage when using SMT, Google proposed a Linux scheduler patch to ensure only trusted tasks share the same physical core in November 2020 [21].

The design of an operating system scheduler emulates near-optimal parallel task execution when the underlying hardware has physical limitations to the amount of true parallelism that can occur. In the case where SMT is used, the scheduler assigns work to the logical CPU cores and relies on features of the underlying hardware to achieve additional parallelization of instructions from different programs.

The Linux kernel uses a symmetric multiprocessing model, meaning that the kernel does not exercise bias towards a particular CPU over against another [4]. In Linux, two basic kinds of scheduling policies exist. There are normal processes (priority 100 to 139) and real-time processes (priority 1 to 99) with lower numbers indicating higher priorities [4]. The default priority is 120 and the nice value is added to the default priority. Table 2 presents a high-level summary of the various scheduling policies available in Linux. The scheduling facility in Linux is modular and custom scheduler implementations are possible.

Since Linux Kernel 2.6.23, the default scheduler is the Completely Fair Scheduler (CFS) that models an ideal scheduler allowing an arbitrary number of tasks to always execute in parallel. CFS replaced the O(1) scheduler [4], which followed a time-slice approach to scheduling and relied on complex heuristics to distinguish interactive processes from batch processes to favor user-initiated tasks [33]. The CFS uses a dynamic virtual runtime metric, rather than a fixed timeslice, to model an ideal scheduler. The amount of time a task will execute on the CPU is a function of the current number of tasks, a task’s nice value, and the minimum time required by the kernel
Table 2. Summary of Linux Scheduling Policies

| Policy         | Real-Time? | Description                                           |
|----------------|------------|-------------------------------------------------------|
| Normal         | No         | Default Completely Fair Scheduler in Linux            |
| Batch          | No         | Assumed CPU-intensive, slightly disfavored            |
| Idle           | No         | Task scheduled as ultra-low priority                  |
| **First in-first out (FIFO)** | Yes | Tasks run until blocked/preempted                     |
| Round Robin    | Yes         | Same as FIFO but fixed execution timeslice            |
| Deadline       | Yes         | Global earliest deadline runs first                   |

Fairness between process groups also exists to prevent a single user from creating many threads to unfairly consume a larger share of CPU resources. When a task is scheduled to the CPU the task will execute for the assigned dynamic timeslice. The actual execution time on the CPU is kept as a virtual runtime statistic. The scheduler maintains a Red-Black tree ordered by the virtual runtime statistic. The leftmost node of the Red-Black tree is guaranteed to be the node with the lowest virtual runtime and will be executed first in a given scheduling period. Tasks that wait for **Input/Output (I/O)** frequently will have lower virtual runtime statistics and will naturally execute ahead of CPU-bound tasks. In practice, this ensures user-interactive tasks are serviced ahead of other tasks scheduled to execute during the run period.

2.4 Other Factors

The previous information is relevant to this research but in practice still more factors influence code execution on real systems. These factors include power management operations performed in hardware to adjust voltage,
operating system power management strategies, and dynamic frequency scaling. Data input and output from lower rate sources further up the hardware architecture stack all contribute to the ways in which programs interact with shared resources. Characterizing the effects of these additional factors are outside the scope of this work.

3 RELATED WORK

The research presented in this article relates to using machine learning for side channel analysis in computer systems. The use of side channels to exploit CPU caches has been an active research area since as early as 2005 when Percival demonstrated the theft of OpenSSL keys using a cache timing attack [28]. Cache timing attacks have been used to break kernel address space layout randomization, a technique to mitigate buffer overflow attacks in modern computers. More recently, Spectre [20] and Meltdown [22] demonstrated the exploitation of microarchitectural weaknesses in cache designs and branch predictors to transgress virtually all security boundaries. The Meltdown vulnerability enabled reading kernel memory from a non-privileged user program. Szefer surveyed microarchitectural vulnerabilities noting that microarchitectural side channels are unique compared to other side channels, because the attacker only needs to run a spy application that is co-located with the victim [35].

Most cache attacks depend on a high-resolution timing source [1, 13, 28]. Computer architecture research, such as the Fuzz runtime environment proposed in Reference [14], considers systems designed to eliminate information leaked using the time domain. These proposals often involve substantial changes to the underlying architecture and carry significant performance tradeoffs making them unlikely to be widely adopted in the near-term.

A common approach to mitigate timing vulnerabilities is to reduce the time source’s resolution. The Google V8 development team used this approach [36] to mitigate the Spectre vulnerability in browsers. In practice, reducing time resolution only mitigates certain types of attacks and Schwarz demonstrated effective techniques to construct and calibrate timers without using built-in function calls [30].

Closely related to side channels are covert channels. This article distinguishes covert channels from side channels in the following way: Side channels unintentionally leak information observable by the attacker, whereas a covert channel transmits information across system security boundaries in a way the system architect did not intend. Masti implemented a thermal covert channel by varying processor workload to increase/decrease processor temperature to send information to a colluding process [25]. The thermal channel exists because the processor consumes more power and thus produces more heat under heavy utilization. Although this is a low bandwidth channel, it affords an opportunity for information to flow through an unintentional communication medium. The interrogator designed for collecting data to support this research was inspired by covert channel communication schemes.

Application inference research is common in mobile device security. Chawla used machine learning to exploit the Dynamic Voltage Frequency Scaling (DVFS) on Android phones to infer running applications [6]. Power conservation is a primary design goal for mobile devices and DVFS operates by increasing or decreasing the processor’s frequency to meet the running application’s demand while satisfying a power consumption policy [15]. DVFS is an effective sensor, because the continuous frequency scaling is likely to follow the inherent structure of a running application.

Global statistics, such as memory and CPU usage were once available on Android through the /procfs [23] file system. When these summary statistics are aggregated they provide a dynamic picture of the current operational state that is correlated to the applications running. Spreitzer developed the ProcHarvester application to scan the /procfs file system for information leaks [34] and then used ProcHarvester to classify 100 applications during startup. The Android operating system now restricts access to most summary statistics available through /procfs [8]. Similarly, Zhang applied machine learning techniques to demonstrate how the iOS API could be
exploited to infer running applications [38]. The nature of mobile devices and their sensor configuration offers a wide surface for machine learning applications.

4 METHODOLOGY

The methodology proposed in this research addresses two distinct areas. First, the methodology establishes an approach to collect structured data using unprivileged processes that is suitable as input for machine learning model development. Second, the methodology will describe a process to select and evaluate an appropriate model to assess how well the developed model recognizes known programs in a set of unseen data that were collected using the same technique. Covert channel research in non-mobile platforms tends to focus on discovering new communication channels, improving channel bandwidth, or using machine learning to detect undesired behavior. Side channel analysis often focuses on physical properties or specific architectural weaknesses (e.g., cache design) to leak data. This research applies the covert channel concepts by approximating the high-level performance statistics used in Reference [34] and Reference [38] through active self-monitoring by an interrogator program. The approach described in this research is fundamentally hard for system designers to eliminate, because it only requires the ability to execute code inside a container on the target system and does not use APIs or system calls.

To answer the research questions posed in Section 1, multiple experiments were devised to execute code in a variety of running configurations and a model search process was used to identify the model that achieved the highest classification accuracy. A set of 10 user programs were chosen as the classification target. For each experiment, the interrogator collected data for the classification task and ran 100 times for each classification target. The upper limit to the number of candidate features considered in this research was 33. One hundred samples was sufficiently large to ensure the training set contained more data samples than features but small enough to fit several different kinds of models while also tuning hyperparameter settings. The data were analyzed with a structured machine learning process to select the model most likely to achieve high classification accuracy on unseen data captured on the target system using the same experiment constraints.

This section discusses five aspects of the methodology: (1) threat model, (2) interrogator design, (3) data collection and preparation, (4) model fitting and selection, and (5) model performance evaluation.

4.1 Threat Model

This research considers a threat model present in containerized execution environments, which are commonly used to provide cloud services. In this container environment, which is implemented using Docker [9], an unprivileged attacker executes her own unprivileged code within the boundaries of a container without being able to effect control flow or data integrity in other containers or the host operating system. System designers may adapt the defender and adversary models described in this section to study sources of information leakage in new production systems or applications.

4.1.1 Defender Capabilities. The container system provider is the primary defender in this threat model and controls the system architecture and container resource allocation. The defender has broad abilities to set container policies, which includes determining the CPU load policy, setting scheduling priorities, and allocating containers to specific CPU cores. The defender is also able to sufficiently constrain an attacker’s container so that neither the ps command, the /proc file system, nor privileged system calls can be used to directly gather information about other activities taking place on the system.

4.1.2 Adversary Capabilities. In this research, the goal of the adversary (Eve) is to design an interrogator program to perform general surveillance by reliably detecting when another user (Alice) executes a specific task with a high level of accuracy. Eve does not control the production environment but she is able to create a development environment with similar system properties to the production environment.
In the approach used in this research, Eve has some prior knowledge about which programs she desires to detect Alice running. Eve also knows the information about the system architecture and the operating system used by the container provider. Eve uses this knowledge to build a system, under her control, where she develops experiments to collect data and build machine learning models to apply on data collected running inside a production container that shares resources with Alice. By maintaining a development environment, Eve is able to study which execution conditions yield the best results.

Figure 5 provides an abstract view of the target operating environment. The applications running in Alice’s and Eve’s containers are oblivious to one another and assume dedicated access to the system hardware resources, which are in fact shared. In the production environment, Eve does not control system resource allocation and therefore cannot determine how much CPU time or memory allocation will be given to her applications.

4.2 Interrogator Program Design

The interrogator program proposed in this research is relatively simple but the principles put forward could be used to devise robust, architecture-specific use-cases. An interrogator program must perform two functions. First, the sensors composing the interrogator program must perform a unit of work. The unit of work could be as small as a single instruction or as large as a sophisticated program. Second, the interrogator must have a sampling capability to count units of work in a given time period. This research uses the ideal case where sampling is performed at fixed-time intervals. However, in environments where timing APIs are obscured, techniques such as a tight for loop incrementing a counter could be used to achieve this goal [30].

The first step in interrogator design is to determine which elements of the computer architecture will be the targets to leak information from. The information leaked in this research is not a specific secret, such as the case with cache side channel research, but rather information about the current system’s operation. However, this research could be applied to further study specific instances of cache side channel information leakage. In a covert channel communication scheme the targets would be the communication channels used to transmit and receive information. This research focused on targets in the main memory system and the on-core execution facilities that are shared when SMT is used. Knowledge about the target system, the distinguishing features in specific applications, and the adversary’s goals are design considerations for the adversary. The memory system is complex and it is possible to target specific layers in the hierarchy such as the L1/L2 cache for information.
The interrogator built for this research was configurable so that it could target various combinations of memory, floating point, and integer execution. These three architectural areas represent a small subset of the possible sources for information leakage. In practice, any resource that must be shared by multiple applications could be a source for leakage. An interrogator program could include measurements for network throughput, disk read/write times, context switching times, and several techniques in the transient execution family to leak information.

Figure 6 introduces a generalized representation of an interrogator program, while Table 3 summarizes the interrogator parameters in this research. The interrogator structure should be flexible to execute a variety of experiments using parameters to establish observation period, sampling rate, and the work units. The sampling thread and each work unit is executed in its own dedicated sensor thread. The work units update a counter variable every time the set of operations is completed. The architectural details introduced in Figure 4 (Section 2) show the number of ALUs, address generation, SIMD and floating point units. When the target context is SMT, the work unit should be constructed to fill the capacity of an execution unit during instruction issue so that delays by competing processes are more likely to occur. The sampling thread uses a read-write lock to ensure competing threads are not executing when the sampling measurement is recorded and the values are reset.

There are many design tradeoffs that can be made developing the interrogator. A low-level performance profiler, such as Intel’s VTune software can provide meaningful information to identify areas for improvement or future analysis. During interrogator design, VTune’s CPU hotspot analysis of the interrogator identified that the interrogator program spent approximately 30% of its execution time spinning to acquire a lock. The time required to acquire a lock could lead to the interrogator failing to sense information. An alternative interrogator design using atomic read/update instructions was considered. The atomic instructions eliminated the software spin time but doubled the number of cycles per instruction, which might have undesirable consequences when trying to learn information about the operating environment. The goal of this research was primarily to demonstrate the effectiveness of an interrogator and not to optimize every property. Therefore, the read-write lock approach was used during data collection, although studying the influence of atomic instructions is of interest.

Figure 7 shows the interrogator program implementation details. This research does not claim the sensor threads are optimal, but that the sensors are sufficient to reveal some information. As a rule of thumb, the sequences of instructions should involve data that are independent from previous instructions to avoid processor stalls waiting for previous instructions to complete. The simplicity of the sensors is intentional and compiler...
optimizations were turned off to prevent optimizations such as loop-unrolling or removal of instructions that do not produce results used elsewhere in the program. An alternate approach could be to implement all instructions using inline assembly. The floating point sensor described in Algorithm 4 uses the sqrt() function. Analysis with the VTune profiler shows that the sqrt() function stresses the floating point divide unit but does not achieve high floating point utilization, because the floating point units are optimized to work with vectorized data. The interrogator was not modified to use floating point vectors, but this functionality could be adopted, especially if a classification target was known to perform many floating point computations on the CPU. The write lock is acquired and held by the sampling thread to synchronize measurement reads and resetting the counters. The read lock is used by the sensors, even though data are modified, because the counter variable is unique to the sensor.

### 4.3 Data Collection

This section describes the data collection process, class labels, and features extracted during pre-processing. Data collection assumes Eve knows which 10 programs Alice uses and has access to the same programs in her experiment environment. Eve acts alone to collect the data by simulating Alice running a program then launching the interrogator. Eve’s goal is to build a model that she is confident will correctly identify Alice’s programs when Alice executes a program in the future.

#### 4.3.1 Operating Environment Parameters

Given the multicore architecture shown in Figure 2 and knowledge about the CFS algorithm, a program’s code can execute on hardware under a variety of conditions. Table 4 summarizes how two concurrently running processes could be executed on real hardware. The hybrid run configuration is the default behavior on Linux systems.

Table 5 defines the experiment parameters that may be set to achieve the run configurations described in Table 4 and other properties related to scheduling and resource management. By default, container users cannot specify the nice values and the container provider must explicitly enable this capability [9].

The data collected during an experiment are influenced by the scheduling parameters described in Table 5, sensor configuration, target hardware, input data, and additional workload present on the system. This wide variety of considerations is a combinatorial problem and only a small subset of possible combinations was considered that would help answer the research questions.

Table 6 lists the properties of the computer systems used to collect data. The range of possible frequencies was initially a source of concern, because slower or faster processor rates could skew the measurements. Dynamic

---

Table 3. Interrogator Parameters Under Adversary Control

| Parameter               | Values   | Description                        |
|-------------------------|----------|------------------------------------|
| Sampling Rate           | 10,000 μs| Time to pause between counter reads|
| Observation Period      | 5,000 ms | Duration to monitor target system  |
| Sensor Thread Target    | Various  | Integer, floating point, or memory operations |
| # of Sensor Threads     | 1..3     | Up to the # of kinds of sensors    |
| Source Code             | C        | C-source code compiled with optimization level 0 [10] |

Table 4. Enumeration of Concurrent Process Execution Mapped to CPU

| Run Configuration | Description                                                                 |
|-------------------|-----------------------------------------------------------------------------|
| Same Logical      | Same logical (thus physical) core.                                           |
| SMT                | Same physical core, but different logical core supported by the same execution units & L1/L2 cache. |
| Different Physical| Different physical cores supported by distinct execution units & L1/L2 cache. L3 cache shared with other cores. |
| Hybrid            | Operating system assigns workload to one of the above configurations but shifts assignment based on system load balancing. |
frequency stepping could have considerable influence on experiments if large changes in frequency occur when the experiments run. To examine the dynamic frequency stepping behavior on the target system, the processor’s frequency was monitored while the interrogator program executed. Within a few ms of starting the operating frequency peaked to the maximum frequency. During execution, the frequency experienced slight steps (within a few MHz up/down) dropping sharply after program termination. Without evidence that the frequency would vary significantly during an observation, the default profiles governing frequency stepping were used.

---

**Algorithm 1 Sampling Thread**

measurements = 16 MB on Heap  
sampling rate = 10 ms

```
write_lock()    # Let all sensors start
sleep(100 ms)
write_unlock()
```

e = current time + observation period

```
while current time < end time do
    write_lock()
    measurements ← count_int, float, mem
    count_int, float, mem = 0
    if measurements is full then
        write measurements to disk
    end if
    sleep(sampling rate)
    write_unlock()
end while
```

measuring = FALSE  
Save measurements to disk

**Algorithm 2 Integer Sensor**

\[ a_0, a_7 = \text{random integers} \]

```
count_int = 0

while measuring == TRUE do
    b_0 = a_0 + a_1
    b_1 = a_1 * a_2
    b_2 = a_0 + a_2
    b_3 = a_0 / a_3
    b_4 = a_4 + a_1
    b_5 = a_4 + a_5
    b_6 = a_5 * a_6
    b_7 = a_6 - a_7
    read_lock()
    count_int += 1
    read_unlock()
end while
```

**Algorithm 3 Memory Sensor**

```
count_mem = 0

while measuring == TRUE do
    ptr = malloc(64K)
    memset(ptr, ‘z’, 64K)
    free(ptr)
    read_lock()
    count_mem += 1
    read_unlock()
end while
```

**Algorithm 4 Floating Point Sensor**

```
arr = 64K random doubles
i = 0

count_float += 1

while measuring == TRUE do
    b = \sqrt{arr[i \mod 64K]}
    i += 1
    read_lock()
    count_float += 1
    read_unlock()
end while
```

---

Fig. 7. Sampler and three sensor thread implementations.

---

**Table 5. Scheduling Policy Parameters**

| Scheduler Configuration | Description |
|-------------------------|-------------|
| Alice’s Core(s)        | Container equivalent of taskset. Determines which cores Alice’s container uses. |
| Eve’s Core(s)          | Determines which cores Eve’s container uses. |
| Alice’s CPU Limit      | Max CPU utilization for Alice’s container. May be \( \geq 1.0 \) if multiple cores available. |
| Eve’s CPU Limit        | Max CPU utilization for Eve’s container. May be \( \geq 1.0 \) if multiple cores available. |
| Alice’s nice value     | The nice value for Alice’s process (default 0). |
| Eve’s nice value       | The nice value for Eve’s process (default 0). |
### Table 6. Target System Configuration

| Component       | Primary System      | Intel Desktop     | Talos II          |
|-----------------|---------------------|-------------------|-------------------|
| System Type     | Laptop              | Desktop           | Desktop           |
| CPU             | Intel i7-4700MQ     | Intel i7-6950X    | IBM POWER9        |
| Physical Cores  | 4                   | 10                | 4                 |
| Threads per Core| 2                   | 2                 | 4                 |
| CPU Max Frequency| 3.4 GHz           | 4.0 GHz           | 3.8 GHz           |
| CPU Min Frequency| 800 MHz            | 1.2 GHz           | 2.166 GHz         |
| L1 Data Cache   | 32 KB               | 32 KB             | 32 KB             |
| L1 Instruction Cache| 32 KB            | 32 KB             | 32 KB             |
| L2 Cache        | 256 KB              | 256 KB            | 512 KB            |
| L3 Cache        | 6 MB                | 25 MB             | 10 MB             |
| RAM             | 16 GB               | 128 GB            | 8 GB              |
| Disk            | 240 GB SATA SSD     | 1 TB SATA SSD     | 100 GB SATA SSD   |
| Operating System| Ubuntu 18.04        | Ubuntu 18.04      | Debian 10.8       |
| Linux Kernel    | 5.4.78              | 4.15.0            | 4.18              |

#### 4.3.2 Experiment Definitions to Address Research Goals

This section describes the experimental parameters used to collect data. Table 7 summarizes the data collection experiments for use when applying the machine learning process described in Section 4.4. Experiment groups are described in Table 8 and were defined to identify experiments related to similar research questions. The set of experiments was derived through a combination of deliberate planning to address the research questions and responding to results in the data. Though this set of experiments does not cover every possible combination of variables it provides a sufficient basis for understanding the strengths and weaknesses of the interrogator and proposed machine learning approach.

Three different levels of background noise were used to create conditions that might negatively impact classification accuracy. At Noise Level 1, four `stress-ng` programs executing tree search, sorting, pipe, and branch stress were run in parallel on the same cores allocated for Eve and Alice. This program ran continuously for the duration of the experiment and represents a fairly constant form of noise. At Noise Level 2, a sequence of 40 benchmarks from Phoronix open benchmarking tool [29], was used to simulate many unknown real-world programs. The sequence of benchmarks ran throughout the duration of the experiment and was restarted from the beginning when all benchmarks completed. The third noise level is the case of an idle system where no background programs (other than core operating system services) were running during data collection. Collecting data and training models this way assumes the adversary has some understanding of background programs that might run alongside the victim.

This research began with a naïve assumption that an interrogator using three sensors would outperform a single sensor. The earliest experiments collected data with this assumption but as it became clearer that this assumption did not hold, a single sensor for the memory unit became the baseline interrogator configuration in experiments.

#### 4.3.3 Description of Alice’s Set of Programs

Ten classes were used, and Table 9 describes Alice’s programs, the sensors that are expected to be useful, possible visibility gaps (i.e., weaknesses in the interrogator design), and the input data. Many of the programs rely on disk I/O operations before work begins but the interrogator does not include a mechanism to observe changes in disk read/write performance. The input data were constant for some programs, though `keygen` and `mysql-bench` have randomness in the behavior. The MySQL server interacting with the MySQL benchmark ran in a different container with CPU cores that were not used to conduct experiments. Given the large number of experiment configurations being considered, the variance due to different input data was not considered in the experiments.

#### 4.3.4 Data Collection and Pre-processing

This section describes how the experiment parameters in Table 7 were used to collect and partition the data for use with machine learning. Algorithm 5 summarizes Eve’s actions...
### Table 7. Experiment Configurations for Data Collection

| ID | Group | Run Config | Float | Int | Alice CPU | Eve CPU | Noise | Processor |
|----|-------|------------|-------|-----|-----------|---------|-------|-----------|
| 1  | A,B   | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 2  | A     | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-6700   |
| 3  | A     | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | IBM POWER9 |
| 4  | B     | Different Physical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 5  | B     | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 6  | B     | Hybrid      | ✓✓✓  | ✓  | 8.0       | 8.0     | -     | i7-4700MQ |
| 7  | C     | SMT         | ✓✓✓  | ✓  | 1.0       | 0.1     | -     | i7-4700MQ |
| 8  | C     | SMT         | ✓✓✓  | ✓  | 0.1       | 1.0     | -     | i7-4700MQ |
| 9  | C     | SMT         | ✓✓✓  | ✓  | 0.1       | 1.0     | -     | i7-4700MQ |
| 10 | D1    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 11 | D1    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 12 | D1    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 13 | D2    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 14 | D2    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 15 | D2    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 16 | D3    | Different Physical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 17 | D3    | Different Physical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 18 | D3    | Different Physical | ✓✓✓  | ✓  | 1.0       | 1.0     | -     | i7-4700MQ |
| 19 | E1    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | 1     | i7-4700MQ |
| 20 | E1    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | 1     | i7-4700MQ |
| 21 | E2    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | 1     | i7-4700MQ |
| 22 | E2    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | 1     | i7-4700MQ |
| 23 | E3    | Same Logical | ✓✓✓  | ✓  | 1.0       | 1.0     | 2     | i7-4700MQ |
| 24 | E3    | SMT         | ✓✓✓  | ✓  | 1.0       | 1.0     | 2     | i7-4700MQ |
| 25 | F1    | SMT         | ✓✓✓  | ✓  | 19        | –20     | –     | i7-4700MQ |
| 26 | F1    | SMT         | ✓✓✓  | ✓  | 19        | –20     | –     | i7-4700MQ |
| 27 | F1    | SMT         | ✓✓✓  | ✓  | 19        | 19      | 1     | i7-4700MQ |
| 28 | F2    | Same Logical | ✓✓✓  | ✓  | 19        | –20     | –     | i7-4700MQ |
| 29 | F2    | Same Logical | ✓✓✓  | ✓  | 19        | –20     | –     | i7-4700MQ |
| 30 | F2    | Same Logical | ✓✓✓  | ✓  | –20       | 19      | 1     | i7-4700MQ |

### Table 8. Descriptions of Experiment Groups

| Group | Description |
|-------|-------------|
| A     | Naïve interrogator with three sensors on the same logical core using different processors. |
| B     | Naïve interrogator from A applied to each run configuration described in Table 4. |
| C     | Naïve interrogator from A, running under container enforced CPU limits. |
| D     | Interrogator with a single sensor for (1) same logical core, (2) SMT, and (3) different physical cores. Data for the hybrid run configuration was not collected. |
| E     | Interrogator running alongside other system workloads in the same logical and SMT run configurations. (1) Memory sensor only. (2) All three sensors. (3) Memory sensor only with a heavy and non-constant workload. |
| F     | Interrogator with memory only sensor running with different nice priorities. (1) SMT Run configuration. (2) Same logical core. |

to collect data in a non-production environment under her control. The execution framework implementing Algorithm 5 ensures that data can be labeled for use with supervised machine learning.

In an experiment, Eve collects measurements for 100 5-second periods on each program, resulting in 1,000 total observations. Within a single observation (i.e., output file) there is a variable number of measurements reported for each sensor in use. The experiment preserved metadata in filenames to extract class labels and sensor configuration during pre-processing. As shown in Figure 6, the interrogator outputs a sequence of work counts as time elapses. The elapsed time in nanoseconds between each sampling measurements is recorded.
Table 9. Alice’s Programs and Data Input

| Program      | Expected Sensors | Visibility Gaps                  | Data Input                                      |
|--------------|------------------|----------------------------------|-----------------------------------------------|
| none         | Any              | Slow I/O                         | Idle System                                    |
| noise        | Any              | Slow I/O                         | Duplicate Interrogator Program                 |
| keygen       | Integer          | Crypto Execution Units           | Private key for 16384-bit RSA pair             |
| bzip2        | Int, Mem         | Compression Execution Units      | Compress 1 GB file                             |
| gzip         | Int, Mem         | Compression Execution Units      | Compress 1 GB file                             |
| ffmpeg       | Int, Float, Mem  | Specialized SIMD Units           | Convert 886 MB .avi file to .mpeg4            |
| jpg2png      | Int, Float, Mem  | Specialized SIMD Units           | Convert 80 MB jpeg file to png                 |
| stress-cpu   | Int              | Single DIV op                    | Single-threaded search for all primes in range 1..1000000 |
| stress-memory| Mem              | Mixing Mem read/write            | Single-threaded stress of read/write 64,32,16 and 8 bit buffer |
| sysbench-mysql| Int, Mem         | Slow I/O                         | 4-threads querying 3 MySQL tables with 1M records each |

ALGORITHM 5: Data Collection Process

1: Run Configuration is Set by Experiment Parameters in Table 7
2: for Program p ∈ {Alice’s Programs} do
  3:   for 1..100 do
  4:     Eve starts p in desired run configuration in Container A
  5:     Eve starts the interrogator in Container B
  6:     Wait 5 seconds
  7:     Save the interrogator data to disk
  8:     Eve terminates p and the interrogator
  9:     Eve waits on p and interrogator process IDs
10:   Eve pauses for two seconds before starting the next interrogation period
11: end for
12: end for

to assess how well the sampling period is honored by the system but was not used as an input feature. The magnitude of integer, floating point, and memory allocation measurements are different due to the number of cycles per operation required by the type of sensor.

Data pre-processing involved extracting labels and computing summary statistics for each execution trace. This research did not investigate classification models using the raw time series. Table 10 summarizes the class labels and the extracted features. Due to various sensor configurations the number of features available can be calculated by multiplying the number of sensors by eleven. The class labels were expressed using a 0..N-1 label encoding scheme for categorical data [12] in the Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Random Forest Classifier (RFC), Decision Tree (DT), and K-Nearest Neighbor (KNN) models. The 10 programs to classify were evenly distributed. The Support Vector Machines (SVM) classifiers converted the labels to a binary representation and used a one-versus-the-rest strategy to make the classification predictions. Some additional preprocessing steps were performed after the resulting data were divided into training, validation, and test datasets.

4.4 Model Development and Selection

This section describes the model development and selection process used to select a high-performing model with scikit-learn [27]. This process was repeated for each experiment to select the model for reporting results. Prior to model fitting, the data were split randomly into three groups while preserving equal distribution of classes.
Table 10. Classes (using Label Encoding) and Features

| Type       | Description                                                                 |
|------------|----------------------------------------------------------------------------|
| Label      | none, noise, keygen, bzip2, gzip, ffmpeg, jpg2png, stress-cpu, stress-memory, sysbench-mysql |
| Feature    | Skewness of the sensor measurement distribution                             |
| Feature    | Kurtosis of the sensor measurement distribution                             |
| Feature    | Mean of the sensor measurement distribution                                 |
| Feature    | Median of the sensor measurement distribution                               |
| Feature    | Standard deviation of the sensor measurement distribution                   |
| Feature    | 25th Percentile of the sensor measurement distribution                      |
| Feature    | 75th Percentile of the sensor measurement distribution                      |
| Feature    | Total number of completed sensor operations                                |
| Feature    | Total number of times the sampler read the sensor’s count                 |
| Feature    | Minimum observed sensor measurement value                                  |
| Feature    | Maximum observed sensor measurement value                                  |

Fig. 8. Distribution of data across training, validation, test sets.

as shown in Figure 8. All model decisions used training and validation data only. After the data were split into train, validation, and test sets, additional steps were taken to create a view of the data with z-scaled features and a view with 10 principal components as features. During model selection K-fold cross-validation (with \( k = 5 \)) estimated the trained model's accuracy. The model with the highest mean classification accuracy during cross-validation was chosen to use when reporting results on the test data. The importance of the validation set is to allow for opportunities to evaluate model performance (beyond cross-validation) and adjust hyperparameters while ensuring test data are never used to develop a model. Confusion matrices for the validation data were used to estimate classification performance and model interpretation.

Model development considered LDA, QDA, RFC, DT, KNN, and SVM as candidate classifiers. For the KNN models \( k \) was fixed at \( k = 4 \) even though multiple parameters for \( k \) could be tried. KNN does not perform feature selection and the time complexity associated with exploring useful features and the best setting for \( k \) was determined to be unreasonable. Support vector classifiers were evaluated using linear, polynomial, and radial basis function kernels [31].

Two approaches described in Reference [18] were used to search for the combination of features yielding the highest cross-validation accuracy during training. The first approach used a forward stepwise procedure to incrementally build up a model from an empty model by adding the single feature to the base model that resulted in the highest accuracy model for a specific iteration. The second approach used the backward stepwise technique to explore the feature space in reverse by removing the feature associated with the worst performing model in an iteration. Neither forward or backwards approaches guarantee an optimal solution but together the techniques offer a reasonable way to explore models containing up to the maximum number of features and perform comparison between the model classes. Additionally, both the forward and backward stepwise approaches allow for interpretation about the relative importance of the feature being added or removed to the classification task.

The RFC does not benefit from the feature selection process in the same way as other algorithms, because random forests inherently perform feature selection. The implementation of random forest classifiers in scikit-learn provides the weights associated with each feature’s importance [32]. Additionally, the RFC does not require features to be scaled, although feature scaling performed slightly better in the experimental results.
The computational complexity conserved by not searching for meaningful features can be used to allow the classifier to increase the number of estimators in the forest. A larger number of estimators will reduce the variance in performance but comes at the cost of more computational time to train a model. The default setting in scikit-learn is 100 estimators, which is a reasonable default setting that was used in this research.

The two approaches were executed on each view of the features (no feature scaling, standardized features, and principal components). During the model search phase with each experiment, all models, lists of features, and cross-validation accuracy were saved for post-processing analysis. Once the model fitting process completes, the model performance metrics were searched to find the best performing and second-best performing model. In the case where two models of the same kind resulted in the same cross-validation accuracy, the bias-variance tradeoff was used as a guide to select the more biased model with fewer parameters. For the experiments in Group A, the data from the different processor architectures were used as training and test data with a new model matching the type of classifier selected by the search procedure. Taking this step provides evidence about whether or not models trained with data from a specific architecture will give high-accuracy results when making predictions for systems with different properties.

4.5 Variance of Results with Different Program Input

To limit the number of factors influencing each experiment and to focus on the run configuration’s influence on the result rather than the input data, the experiments were executed with the same input data. Real-world applications might be concerned with how the approach described in this article performs when Alice’s programs are exposed to different inputs. In general, the more Eve can vary program input while collecting training data, the more she will be able to reduce variance in the results.

To demonstrate this principle, the results from Experiment 15, which achieved 100% classification accuracy, will be augmented with a new small dataset. There are 25 samples for each of the 10 program classes meaning there are 250 total samples in this dataset. The new dataset was created by varying the source image, data to compress, secret key, and number of threads for the stress-ng programs. Two different tests were performed to explore the influence of variation in data input. First, measure the accuracy of the original RFC on all the new 250 samples without any additional training. Second, partition the test set into 150 samples for training and 100 samples for test, maintaining even class distribution. Fit a new RFC using the original 500 samples plus the new 150 training samples. Test the new model on the 100 samples reserved for a test set. The results for this analysis are reported in Section 5.7.

The first test results will likely yield a poor result, because the model has experienced only minimal variability in program input. In the second set of test results, the results should improve considerably, since the model fitting algorithm will be exposed to more diversity in the training process. Confusion matrices will be used to demonstrate the differences in model performance.

4.6 Useful Investigative Plots for Analysis

In a classification problem the accuracy depends on the degree to which data are separable. This section presents examples of exploratory plots that can be used on the training data prior to an experiment to understand how machine learning models might perform. These plots provide insight based on the input data and are not machine learning models.

4.6.1 Violin Plots of Raw Sensor Values. The distribution of magnitudes within the raw time series is important. The features used for machine learning are summary properties of this distribution and therefore the absence of differences within these plots provides information about the degree to which a particular sensor reveals additional information in an experiment.

Figure 9(a) shows that Alice’s running program affects the number of times Eve’s interrogator program is able to sample in a five second period. Figure 9(b), (c), and (d) show that the different distributions of work counts...
for a three-sensor interrogator when background noise is present. The features are min-max scaled within the sensor type and there does appear to be uniqueness to the distribution across all three sensor types, suggesting multiple sensors will yield better classification accuracy in a noisy collection environment.

4.6.2 Principal Components Analysis (PCA). Principal Components Analysis (PCA) is a useful technique for reducing the dimensionality of data even though others have demonstrated how autoencoder neural networks can improve dimensionality reduction [16]. The principal components are a low-dimensional representation of the data that contains most of the variation [18]. The principal components include influence from all available model features. The first two principal components can be plotted against one another to see if clustering behavior occurs. The clusters forming in Figure 10(a) show strong indications that the data will be separable for most of the machine learning models considered in Experiment 15 while Figure 10(b) shows separability may be achievable for sysbench-mysql and no-noise in Experiment 1 but the other classes are difficult to separate visually.

4.6.3 t-Stochastic Neighbor Embedding (t-SNE). t-Stochastic Neighbor Embedding (t-SNE) is a visualization technique for high-dimensional data that reduces the tendency of points to cluster together [37]. Figure 11(a) demonstrates how t-SNE can be used to further distinguish the decision groups when compared to Figure 10(a). Although Figure 11(b) does not make the task of visually separating the data from Experiment 1 simple, it shows an improvement for at least some of the program classes.

4.7 Model Evaluation Details

Once the final model for an experiment is selected, three evaluation criteria were considered to interpret the model’s implications related to the current experiment group. The first criteria was mean cross-validation score used to rank the model’s output by the forward stepwise procedure. The second criteria was the confusion matrix for the chosen model on the validation set. The third criteria was the F1-score, which represents a balanced view...
of the model’s precision and recall. If required, then detailed tables of by-class precision and recall can be used to determine which programs are difficult to classify and which type of error is present in the model. Finally, the chosen model’s performance is measured using test data.

5 RESULTS
This section presents cross-validation results from model selection and performance with test data. This section concludes with a reflection on the hypotheses and interpretation of the results related to Eve’s adversarial goals.

5.1 Group A: Different Processors
In Group A, the same logical run configuration was used to collect data for three different processors where all processes had equal priority and no background noise beyond operating system services was present. The results shown in Table 11 provide evidence that the approach in this research is applicable to different computer systems. A natural question associated with this group of data is “How well does the best performing model perform when applied to data from a different architecture?” The classifier chosen for Experiment 1 was tested against the data from Experiments 2 and 3 achieved a classification accuracy of 0.239 and 0.230, respectively. Training a new RFC with data from Experiments 1 and 2 while testing against Experiment 3 data yielded similarly poor results.
Table 11. Summary of Group A Results for Classifier Performance

| ID | Float | Int | Memory | Processor | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|----|-------|-----|--------|-----------|------------|----------------|-------------|----------|--------|
| 1  | ✓     | ✓   | ✓      | 17-4700MQ | RFC        | ✓              | 0.8420      | 0.8440   | 0.8403 |
| 2  | ✓     | ✓   | ✓      | 17-6700   | RFC        | ✓              | 0.9320      | 0.8880   | 0.8883 |
| 3  | ✓     | ✓   | ✓      | IBM POWER9| RFC        | ✓              | 0.8660      | 0.8920   | 0.8895 |

Table 12. Summary of Group B Results for Classifier Performance

| ID | Run Config | Float | Int | Memory | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|----|------------|-------|-----|--------|------------|----------------|-------------|----------|--------|
| 1  | Same Logical | ✓     | ✓   | ✓      | RFC        | ✓              | 0.8420      | 0.8440   | 0.8403 |
| 4  | Different Physical | ✓ | ✓ | ✓ | QDA | ✓ | 0.8080 | 0.7720 | 0.7695 |
| 5  | SMT        | ✓     | ✓   | ✓      | QDA        | ✓              | 0.9900      | 0.9800   | 0.9797 |
| 6  | Hybrid     | ✓     | ✓   | ✓      | RFC        | —              | 0.7560      | 0.7120   | 0.7115 |

Table 13. Summary of Group C Results for Classifier Performance

| ID | Float | Int | Memory | Alice CPU | Eve CPU | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|----|-------|-----|--------|-----------|---------|------------|----------------|-------------|----------|--------|
| 7  | ✓     | ✓   | ✓      | 1.0       | 0.1     | RFC        | ✓              | 0.9560      | 0.9560   | 0.9554 |
| 8  | ✓     | ✓   | ✓      | 0.1       | 1.0     | LDA        | ✓              | 0.7040      | 0.6640   | 0.6651 |
| 9  | ✓     | ✓   | ✓      | 0.1       | 0.1     | RFC        | ✓              | 0.6700      | 0.6640   | 0.6612 |

Training the RFC with the training sets from all three experiments produced a model achieving an accuracy of 0.860 on the combined test sets. This result points to the importance of obtaining data that is representative of the target environment where the model will be applied.

5.2 Group B: Varied Run Configurations

For Group B and the remainder of the experiments, the processor was held constant to reduce the number of data collection experiments. The experiments evaluated in this group were conducted on a running system where the only background noise were standard operating system tasks. Table 12 summarizes the results of the model search process. Notably, code executing in the SMT configuration achieves an near-perfect classification accuracy score of 0.9800 on unseen test data. Given the nature of how transient execution vulnerabilities exploit SMT in many situations, the results here should be expected. Perhaps more significant, though is the realization that moving work to different physical cores does not make the classification task impossible. In all run configurations, the models performed significantly better than random chance.

5.3 Group C: Container Enforced CPU Limits

The experiments in Group C used the SMT run configuration and were chosen to consider the impacts to classification accuracy if a security engineer used maximum CPU utilization policies to mitigate information leakage against the proposed threat. The results in Table 13 suggest an adversary does not need to have high CPU utilization to apply this technique effectively. Even with a 10% maximum CPU utilization Eve is able to achieve a high classification accuracy of 0.9560 on test data. Perhaps a more effective strategy is to limit the victim’s CPU utilization causing Eve to observe more noise than actual target program behavior. Security engineers may use this insight to inform load balancing policies.

5.4 Group D: Single Sensors for Three Run Configurations

The primary purpose for the data collected in Group D was to re-evaluate the early assumption that multiple sensor threads would lead to more data and better performing classifiers. Table 14 shows that each sensor has substantial ability to reveal information about other processes but the memory sensor tends to yield the highest mean cross-validation accuracy.
Table 14. Summary of Group D Results for Classifier Performance

| ID  | Run Config  | Float | Int | Memory | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|-----|-------------|-------|-----|--------|------------|----------------|-------------|----------|--------|
| 10  | Same Logical | ✓     | —   | —      | RFC        | ✓              | 0.7740      | 0.7120   | 0.7015 |
| 11  | Same Logical | —     | ✓   | —      | RFC        | —              | 0.7340      | 0.7400   | 0.7356 |
| 12  | Same Logical | —     | —   | ✓      | RFC        | ✓              | 0.9000      | 0.8440   | 0.8440 |
| 13  | SMT          | ✓     | —   | —      | QDA        | ✓              | 0.9920      | 0.9880   | 0.9881 |
| 14  | SMT          | —     | ✓   | —      | RFC        | ✓              | 0.9980      | 1.0000   | 1.0000 |
| 15  | SMT          | —     | —   | ✓      | QDA        | ✓              | 0.9980      | 1.0000   | 1.0000 |

Table 15. Summary of Group E Results for Classifier Performance

| ID  | Run Config  | Float | Int | Memory | Noise | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|-----|-------------|-------|-----|--------|-------|------------|----------------|-------------|----------|--------|
| 19  | Same Logical | —     | ✓   | —      | 1     | RFC        | ✓              | 0.6300      | 0.5840   | 0.5733 |
| 20  | SMT          | —     | ✓   | 1      | QDA   | ✓          | ✓              | 0.8580      | 0.8880   | 0.8860 |
| 21  | Same Logical | ✓     | ✓   | ✓      | 1     | DT         | ✓              | 0.4900      | 0.4640   | 0.4634 |
| 22  | SMT          | ✓     | ✓   | ✓      | 1     | QDA        | ✓              | 0.8760      | 0.8680   | 0.8686 |
| 23  | Same Logical | —     | —   | ✓      | 2     | RFC        | ✓              | 0.5460      | 0.4840   | 0.4790 |
| 24  | SMT          | —     | —   | ✓      | 2     | RFC        | —              | 0.5440      | 0.5440   | 0.5448 |

Care should be taken when interpreting these results. Recall from Section 2.2 the complexity of multitasking computer systems. Isolating components to assert the measurement is from one specific activity, without elevated privileges is difficult (if not impossible). For example, the count variable, which is read from the sampling thread, resides in memory shared between the threads and therefore every integer count likely captures some effect related to the memory system’s state. Depending on her goal, an adversary may prefer to use only an integer sensor when conditions are known to involve the possibility for SMT, because the effects rely on a single variable in memory versus continuously allocating and freeing memory in a way that might be detected by a defender.

5.5 Group E: Effectiveness with Competing Workloads

The results in Group A through Group D were obtained under conditions favorable to an adversary. Often conditions are not favorable to an adversary. This research sought to provide a sense for the classification accuracy that could be expected when the interrogator collects data under less-than-ideal circumstances. Table 15 shows that noise, particularly heavy and non-constant noise, significantly reduces the classification accuracy. Recalling how CFS calculates the scheduling period, the benchmarks may contribute to this result by spawning a sufficient number of threads to increase the scheduling period so the sampling rate cannot be achieved. Although the classification accuracy experienced a significant decline, the utility of the approach depends on the adversary’s objective.

One possible explanation for the sharp decrease in accuracy is that under the heavy system load created by the benchmarking application the reliability of the sampling thread to take measurements on time suffers. Figure 12 shows the distributions of the time lapse between two sampling measurements for a single run of the bzip2 program at each of the different noise levels. To leverage this information, manipulation of the original time series data would be required. This research focused on machine learning using only the summary properties of the distribution of observed values over time. Therefore, further data manipulation to deal with unreliable sampling behavior was not performed.

The t-SNE plots for Experiments 23 and 24, which had the worst set of classification accuracy results, are shown in Figure 13. Both the same logical and the SMT run configurations performed at about the same level suggesting the Phoronix benchmark suite adds a degree of noise that may be difficult to separate for any machine learning algorithm.
The confusion matrices can be used to diagnose how classification performance fails. Figures 14 and 15 compare confusion matrices for the validation data from the set of experiments without additional background programs running versus the corollary experiment at Noise Level 2. A strong diagonal indicates the classifier achieves high classification accuracy. If the algorithm were performing near random chance, then the diagonal would have approximately $\frac{1}{\# \text{ of classes}}$ in each diagonal position.

Interestingly, the diagonal in both pairs of confusion matrices indicates slightly better than random chance performance under noisy conditions. There also seem to be clusters within the confusion matrices where programs might exhibit similar behavior that is less distinguishable in the presence of system noise. The accuracy for these results are increased due to the strong classification accuracy of a subset of programs. The no-noise class (idle state) and sysbench-mysql were accurately classified in both the same logical and SMT run configurations with background noise.

5.6 Group F: nice Influence

Table 16 provides insight about the influence of the nice values used alongside the CFS algorithm. This experiment used only the memory sensor and considered two of the four run configurations. The value $-20$ is the

Digital Threats: Research and Practice, Vol. 4, No. 2, Article 17. Publication date: August 2023.
highest priority a non-real-time process can hold, while the value 19 is the lowest priority. The results reported in Table 16 suggest the nice value only slightly reduces a classifier's accuracy when an interrogator program is used.

5.7 Influence of Varying Input Data
This section uses the resulting model and data from Experiment 15 (Section 5.4) but evaluates performance of the original model on new data. These results demonstrate the importance of data diversity in the model training.

Fig. 14. Confusion matrices for same logical core without noise and with noise.

Fig. 15. Confusion matrices for SMT without noise and with noise.
### Table 16. Summary of Group F Results for Classifier Performance

| ID | Run Config | Alice nice | Eve nice | Noise | Classifier | Feature Scaling | Mean CV Acc | Test Acc | Test F1 |
|----|------------|------------|----------|-------|------------|----------------|-------------|----------|--------|
| 25 | SMT        | 19         | −20      | −     | DT         | ✓              | 0.9940      | 0.9840   | 0.9840 |
| 26 | SMT        | −20        | 19       | −     | RFC        | ✓              | 0.9980      | 0.9800   | 0.9800 |
| 27 | Same Logical | 19         | −20      | 1     | RFC        | ✓              | 0.9600      | 0.9400   | 0.9471 |
| 28 | Same Logical | −20        | 19       | −     | RFC        | −              | 0.8360      | 0.8560   | 0.8539 |
| 29 | Same Logical | −20        | 19       | −     | RFC        | ✓              | 0.8400      | 0.8360   | 0.8356 |
| 30 | Same Logical | −20        | 19       | 1     | RFC        | ✓              | 0.5513      | 0.5378   | 0.5288 |

Fig. 16. Confusion matrices for different training conditions.

The original model reported only a 65% classification accuracy on the new data and Figure 16(a) shows a significant portion of the new data being incorrectly classified as `sysbench-mysql`. By incorporating a little bit of new data into the training process, a new model achieved 84% classification accuracy on test data. Figure 16(b) shows that exposure to new data does not completely eliminate the poor performance for `jpg2png` or `keygen`. The features used in this research were extracted from the distribution of values over a five second period. Although dealing with the time series nature of the data was outside the scope of this work, additional features could be extracted by dividing the five second period into several smaller intervals and calculating local statistics for each of the shorter intervals. Summarizing the series rather than using the data directly means that the features may only be accurate when applied to an observation of equal duration.

5.8 General Comments on Model Performance and Important Features

This section briefly discusses the general trends in model performance. In most cases, the RFC achieved the highest classification accuracy, which is likely related to the way that random forests build and decorrelate the tree [18]. QDA also yielded a few high performing models, but even in these cases the RFC performed almost equivalently and the RFC was not chosen because of one or two mispredictions. Somewhat surprising was the fact that none of the SVM classifiers were chosen as the best performing model. The SVM with the linear kernel tended to outperform the SVMs with polynomial and radial basis function kernels. The SVMs reported reasonable results that were within 5% of the best performing classifier for some experiments. The SVM models that
performed better during cross-validation typically used all the available features that could have led to slight model overfitting that was manifested in lower test performance.

The forward stepwise procedure and RFC provide mechanisms to understand which features the type of model considered important. In the case of the forward stepwise procedure, the order in which features are added to the model indicates that feature’s relative importance given the information already available to the model. When three sensors were used simultaneously in the same logical core run configuration (Experiment 1), at least one of the memory features was consistently the first feature added to the model. This result fits with the expectation that the operating system does not schedule Alice and Eve for execution at the same time. However, the memory system is shared between the two programs in the same logical core configuration and therefore leaks the most information. Interestingly, in Experiment 5 when all three sensors were used in the SMT run configuration an integer feature was the first feature added for all five classifiers. The change in features between the SMT and same logical configurations indicates a lower-level interaction in hardware between Alice and Eve’s programs.

The RFC calculates an individual feature’s relative importance, which can be compared with the stepwise results. Figure 17 shows a heat map with the relative feature importance, aggregated across all features for an execution unit for Experiment 1 and Experiment 5. This provides a concise visual representation that supports the inference made using the forward stepwise procedure for Experiments 1 and 5.

6 CONCLUSION AND FUTURE WORK

This research demonstrated how an active interrogator can collect data to develop machine learning algorithms that are capable of achieving up to 100% classification accuracy under certain execution conditions. This research described a process to search for different models that achieve high classification accuracy and provided results under a variety of system run configurations. The results were obtained using Linux containers intended to provide strong guarantees of code isolation. Even in this constrained environment the adversary achieved her goals without using system commands such as ps nor relying on restricted system APIs. This section reviews the research hypotheses in the context of the reported results and presents considerations for future work.

6.1 Evaluation of Hypotheses

The goal of this research was to demonstrate how an adversary could use knowledge about a victim’s operating environment to train machine learning models to reliably detect and classify which programs run in the victim’s environment. The utility of this approach ultimately depends upon the adversary’s prior knowledge of the target and the intended use case. However, this form of adversarial thinking could also be beneficial to system architects seeking to develop adversarial models for their systems to evaluate possible information leakage.

The first research question dealt with the kinds of statistical machine learning models that would yield the highest classification accuracy. In general, the RFC outperformed other models considered in this research and
provides insight about important features without executing the expensive forward and backward stepwise procedures. An adversary could choose to use only the RFC model and forgo spending time searching and tuning models, yet still achieve high-accuracy results.

The results described in Section 5 provide a compelling answer to the second research question regarding the execution conditions that allow Eve to accurately detect which of Alice’s programs was executed. The results demonstrate the feasibility of achieving perfect detection accuracy when an adversary executes code on the same physical core under SMT conditions without background noise. The SMT results underscore the importance of Google’s proposed Linux Kernel [21] patch that prevents code from different trust levels sharing a core. However, the results presented in Section 5.2 and 5.4 demonstrate the feasibility of developing a model that accurately identifies one of Alice’s programs at least 70% of the time under all of the possible run configurations. Systems that try to eliminate information leakage by preventing the SMT run configuration may not be sufficient. The execution environment property exerting the most negative influence on Eve's ability to reliably identify Alice’s program was the system noise contributed by the benchmark suite. This suggests that a system might be able to reduce information leakage by maximizing the quantity of work performed by the system.

6.2 Future Work

The research presented in this article could be extended in several ways. First, the interrogator approach could be expanded to perform online detection to study how an adversary could use these techniques to detect system state changes in real time. The time series nature of the data could be combined with neural network architectures designed to learn from sequences to make fast online predictions.

Second, this research considered sensors designed to leak information from common execution units dealing with integer, floating point, and memory. The potential for information leakage by SIMD instructions was not considered, but these specialized instructions could provide valuable information for certain types of victim programs. Similarly, this research focused on the Intel architecture but architectures such as the IBM POWER9 use co-processors for compression and encryption. These co-processors offer another type of on-chip facility to consider when evaluating information leakage in a system.

Third, this research only considered the condition when Alice runs a single program. In practice, Alice is likely to have multiple programs or containers executing simultaneously. Future research could consider techniques that are capable of predicting multiple programs from the data a single interrogator produces.

Finally, this research points to high levels of competing program noise as a primary mechanism to control information leakage in systems. However, adding noise has performance consequences. Developing a better framework to characterize the noise and determine both an appropriate level of background noise and at what points in time the added noise is most beneficial was outside the scope of this research. In the context of container technology, the container provider could offer noise-introduction as a configurable security parameter. Characterizing the proper levels of operating system noise that balance performance and conceal information is an important area of research that merits further study.

In conclusion, this article presented an interrogator program to collect leaked information from three different execution units under a variety of code execution configurations. The information collected by the interrogator program was used to develop a machine learning model that could identify one of 10 programs with 100% accuracy under simultaneous multithreading conditions. An adversary could use machine learning models such as these to collect information about other users across container provided boundaries to achieve surveillance goals. From a defense perspective, system designers could use this research as an approach to study the risks of information leakage from a program or an execution environment.

REFERENCES

[1] Onur Acicmez, 2007. Yet Another MicroArchitectural Attack: Exploiting I-Cache. Technical Report 164. Samsung Information Systems America, Samsung Electronics, San Jose, CA.
[30] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In Financial Cryptography and Data Security, Aggelos Kiayias (Ed.). Vol. 10322. Springer International Publishing, Cham, 247–267. https://doi.org/10.1007/978-3-319-70972-7_13
[31] scikit learn. 2020. C-Support Vector Classification. Retrieved June 9, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.
[32] scikit learn. 2020. Random Forest Classifier. Retrieved June 9, 2020 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.
[33] Chandandeep Singh Pabla. 2009. Completely Fair Scheduler. Linux Journal. Retrieved February 8, 2021 from https://www.linuxjournal.com/node/10267.
[34] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully automated analysis of Procfs side-channel leaks on Android. In Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS’18). Association for Computing Machinery, New York, NY, 749–763. https://doi.org/10.1145/3196494.3196510
[35] Jakub Szefer. 2019. Survey of microarchitectural side and covert channels, attacks, and defenses. J. Hardw. Syst. Secur. 3, 3 (2019), 219–234.
[36] Ben L. Titzer and Jaroslav Sevcik. 2019. A year with Spectre: A V8 perspective. V8. Retrieved August 12, 2020 from https://v8.dev/blog/spectre.
[37] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using T-SNE. J. Mach. Learn. Res. 9, 11 (2008), 2579–2605.
[38] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In Proceedings of the Symposium on Network and Distributed System Security. 1–15.

Received 31 August 2020; revised 18 March 2021; accepted 29 April 2021