Fingerprinting Communication and Computation on HPC Machines

Sean Peisert
Computational Research Division, Lawrence Berkeley National Laboratory
Department of Computer Science, University of California, Davis
sppeisert@lbl.gov / peisert@cs.ucdavis.edu

June 17, 2010

Abstract

How do we identify what is actually running on high-performance computing systems? Names of binaries, dynamic libraries loaded, or other elements in a submission to a batch queue can give clues, but binary names can be changed, and libraries provide limited insight and resolution on the code being run. In this paper, we present a method for "fingerprinting" code running on HPC machines using elements of communication and computation. We then discuss how that fingerprint can be used to determine if the code is consistent with certain other types of codes, what a user usually runs, or what the user requested an allocation to do. In some cases, our techniques enable us to fingerprint HPC codes using runtime MPI data with a high degree of accuracy.

1 Introduction

At supercomputing facilities worldwide, large-scale scientific computation happens as follows: a user applies for "computing time" on a system, the user is given a time allotment, along with access to a login node and the ability to submit to a batch queue. Sometime later, depending on the resources requested and available, the requested job runs on the high-performance computing (HPC) system, and eventually finishes. But what was the computation actually doing? Was it doing what the user applied to do, or something else? Alternatively, is it possible that an authorized user was misusing resources, or perhaps that a non-authorized user gained access to an authorized user's credentials?

We have developed techniques to classify behavior of parallel computations by analyzing application-level behavior. Specifically, we have developed methods that enable us to analyze communication patterns and computational resources used by the applications running on HPC machines. We can then map those patterns either to what the user applied for an allocation to do, to the "normal" use of the user, or to some atomic notion of computation to the types of computation used by known applications. The distinction between the first two concepts and the third is somewhat akin to the concept of *intrusion detection* [Den87] vs. *computer forensics* [PBKM05] in computer security: the former seeks to know *if* something occurred, and the latter seeks to know *what* occurred. Ideally, both could be automated. However, they do not have to be, nor are these processes mutually exclusive. In the absence of automated (or real-time) forensic techniques, detection may simply be an automated alert that triggers a manual analysis.

Specifically, we have sought to answer the questions:

1. Is a particular job similar or different to the jobs a user normally runs?
2. Is a particular job similar to the kinds of jobs that a user would be running based on what they applied for computing time to run?

3. Does a particular job bear a resemblance to a particular set of computations that the HPC system should not be used for?

The challenge in this scenario is to develop a means of distinguishing runs of computation so that different implementations of the same algorithm can be distinguished from entirely different algorithms. Or, alternatively, can we distinguish between the same algorithm running different datasets? If we cannot determine an program from its signature, might we be able to determine what set of codes it is *not*?

The approach that we ultimately use in this paper is to analyze communication patterns by using data generated from IPM (Integrated Performance Monitoring) [BCO+05]. IPM can be configured to monitor a variety of runtime events, including MPI (Message Passing Interface) library calls, POSIX I/O calls, and others. We primarily focus on the MPI calls in our work. We analyze these through a variety of machine learning techniques, that we describe later.

In the rest of this paper, we discuss related work, the data available, a variety of possible techniques for analysis, our actual analytical approach, and finally, our results and conclusions.

2 Related Work

Traditionally in computer security, there are two kinds of analysis performed: static or dynamic (occurring at runtime). Static analysis [Bis03, §23] typically involves analyzing source code or the state of the operating system, disk, or configuration files and performing some kind of property-based testing [FB97], model checking [CDW04], or automated theorem proving [LMW+00]. For example, if an invariant is that a thread needs to enforce a "lock" in a section of code before a shared variable is used (to check for certain kinds of race conditions), then a model checker can analyze source code to see if this is true. Similarly, automated theorem proving can look to see if a particular formula holds for all possible models.

Dynamic analysis [Bis03, §25], typically *intrusion detection* [Den87], involves looking at events at execution time. In computer security, those events are often streams of network packets [MHL94], system calls [Hof99], library calls [PBKM07], and "syslog" events. Such events are used for network or host-based intrusion detection [Bac00]. The analyses can be via statistical variance from the norm (*anomaly detection*) [JV91] or via a set of data that matches a pre-defined pattern (*specification-based* or *misuse-based* intrusion detection) [KFL94, Pax99, ZHR+07]. Anomaly detection has been successful in limited, focused applications. In larger or broader applications, such as general purpose intrusion detection, anomaly detection has largely failed [GT06, SP10]. There are a number of reasons for this: tuning the parameters that control the alert thresholds and control the number of false positive or negative errors is hard; untainted training data is difficult to obtain, and the base-rate difficult to determine [Axe00]; intruders can "game" the system over time; and the cost of errors is high – every false positive is wasted time for security administrators. Finally, anomaly detection systems typically do not indicate *what* was anomalous, just that something was different.

Anomaly detection systems are used in practice, but typically in a way that sends alerts to human system administrators, and not in a way that actually blocks access to a resource. Deterministic, signature-based methods, such as misuse or specification-based intrusion detection are typically reserved for blocking. On the other hand, anomaly detection systems can be useful indicators for further forensic investigation to determine what happend [PBKM07, Pei07].

Most intrusion detection has focused on network packets or system-level events. Very few intrusion detection systems (IDSs) focus on application-level events. Those that do traditionally focus only on database security [Den87] because very few other applications are worth monitoring:

most damage is done to computer systems through system calls, and so one might as well just monitor via kernel events rather than library calls and other events in userspace.

The task of monitoring supercomputers is different for several reasons. First, there is more than one machine in question—there could be thousands of processors. Second, the networks are extremely difficult or impossible to monitor in real-time without affecting performance. Finally, perhaps more than in any other instance, the users of the machines are so sensitive to any performance loss, because they are paying explicitly for a large, fast computing resource. However, HPC systems often already collect relevant data about communication and computation, for use in analyzing performance, that has negligible impact on performance. Thus, the obvious source of data to use for analyzing programs on supercomputers is what is already captured regarding computation and communication: the two primary tasks of the machine.

We believe that both static and dynamic analysis techniques can be applied to the question of analyzing programs on supercomputers. Both methods have been successful in other areas, and both have advantages, though neither is obviously superior. When used in tandem, classification accuracy is generally increased. For the time being, we use the run-time technique, but will pursue combining that technique with static analysis in the future to examine any improvements in accuracy.

There has been a limited amount of existing research on analyzing HPC communication patterns for the purpose of security or fingerprinting. For example, some work has been done to analyze statistics regarding communication of the NAS Parallel Benchmarks [BBB+91] and other HPC programs, but did not look at the actual patterns of communication [FY02, VM03]. Other researchers have analyzed communication patterns in detail, but did not attempt to fingerprint those patterns or apply them to pattern matching heuristics [SKOS05, Rie06, KOPS09, HKB+09].

In the closest work to our own, researchers at Sun Microsystems's Asia Pacific Science and Technology Center in Singapore describe a method for distinguishing among four of the NAS parallel benchmarks using IPM data [MTM+09]. The work is successful, albeit preliminary. For example, it focuses only on four NAS benchmarks, and no other codes. It also does not consider collective communication: only point-to-point communication such as `MPI_Send` and `MPI_Recv` function calls. This work informs our own approach, but the fact that it is still preliminary (even using only four NAS parallel benchmarks) leaves a number of open questions. For example, any small dataset can be easily fingerprinted, but what happens when unknown code is run? One particularly useful approach from this earlier work is the method of *rank transformation* used, allowing runs of different numbers of processors to be compared without simply truncating the larger run of processors to the smaller number of processors in order to compare the two.

3 Approach

3.1 Data

To perform a run-time analysis, one needs run-time data. On the NERSC supercomputers, the most obvious runtime data regarding codes that is already being collected is IPM (Integrated Performance Monitoring) data, principally containing information on MPI use. The level of MPI data obtained varies proportionally to the impact on machine performance: the more data, the greater the slowdown of the code.

There are five levels of IPM data available, in order from most to least information, and also in order from the most to least computationally intensive to collect:

1. Full traces of ordered IPM logs (MPI and POSIX I/O calls), with each processor synchronized as much as possible to avoid non-determinism, to show not only order of calling within an

individual node but order of calling relative to the other nodes.

2. Full traces of ordered IPM logs (MPI and POSIX I/O calls), with each processor unsynchronized, so that only order within each node is known.

3. Aggregated IPM logs checkpointed frequently so that even if ordered traces cannot be captured, one might have some means of determining where one "dwarf" ends and another begins during a long run.

4. Aggregated IPM logs. Aggregated summaries of the communication data used when the setting of `IPM_REPORT=full` is given, for performance reasons. Shows the names of the calls made, the nodes that made the calls, the number of times those calls are made, the names of the users who submit the jobs, and the total number of bytes sent per those calls.

5. Aggregated IPM logs that show the names of calls made, the number of times those calls are made, the names of the users who submit the jobs, and the total number of bytes sent per those calls, but does not show any information about which processors send and receive those calls. This is the output when the setting of `IPM_REPORT=terse` is used.

In each case, we are also able to capture data from up to four hardware counters, which may say something about the nature of the *computation* not just *communication*. For example, we could capture something regarding the number of floating-point operations completed.

It is an open research question as to the level of data actually necessary to perform accurate pattern matching between different programs running on HPC systems. Can we use less data if we use the approach of comparing a particular computational job to a particular user's normal behavior, it is likely that less data might be necessary than if we actually try to identify the specific computation running. In the latter case we might do this, for example, by identifying *computational dwarves* [ABC+06] or some other reasonable proxy, like NAS Parallel Benchmarks. We could then use those to determine whether a particular program known to use those dwarves is running. However, at some point, do "summaries" make the problem impossible because code that involves running multiple "dwarves" will aggregate the dwarves together in a manner that is probably irreversible, and so individual dwarves will be unrecognizable? For example, if multiple dwarves exist in a particular code, and each has a different adjacency matrix, when the matrix for each dwarf is aggregated together, are the more subtle dwarves, performing less communication, masked by the dwarves communicating more, and with more nodes? This is also an open research question, so we will ultimately look at both options in our future work.

In the analysis discussed in this paper, we use IPM data that is collected in an aggregated fashion (#4). The fields in this data include: "from" node, "to" node, call name (e.g., `MPI_Send`, `MPI_Scatter`, `MPI_Bcast`, etc...), bytes sent per call, IPM region, and number of times the call is made. (Of course, with collective calls, the "to" node is irrelevant.) Thus, for MPI calls, we have a four or five-dimensional matrix. The size of the datasets can range from a few megabytes to many hundreds or gigabytes of megabytes for long runs using thousands of processors. Our datasets include twelve distinct runs of aggregated logs using a mixture of 64 and 256 processors per run, and eight of which was run with both 64 and 256 processors per run.

3.2 Method

The techniques involved in analyzing ordered traces are similar to techniques to analyze unordered, aggregated logs. Both use variations on outlier detection, but analyzing the unordered logs primarily relies on analysis adjacency matrices [SKOS05, KOPS09], whereas ordered data contains the additional feature of which calls from/to which nodes tend to happen in which order.

In general, it is very hard to predict which features will be most useful for a given classification problem. Analyzing the matrices alone is one possibility, but what does it mean for one matrix to be *similar* to another matrix? Subtle variations may or may not indicate substantial differences. Additionally, how should scaling be accounted for? For example, if the number of processors used in a job is twice the number of a separate run of the same algorithm, should one scale the results by halving results? Finally, after accounting for the scaling as a result of different numbers of processors used, how should the size of the input, or the tuning options be accounted for?

Another possibility that avoids many of the scaling issues is to guess or derive what data to feed in a way that is largely independent of the number of processors involved. Intuitively, one might guess that things to look at include the average degree of each node, the amount of data sent per node as a ratio with the total amount of data, the ratio of collective to individual communication, etc... In this paper, we first attempt some straight forward and simple features to explore the parameter space.

However, it is also clear that the "features" (e.g., ratios of individual to collective communication, "degree" of nodes) that we are using so far are insufficient, because even in these early analyses, there are still significant variations in the results. It is clear that we need additional insight about relevant "features" in the data. Often this initial technique of beginning with the most high-level features solves a significant part of the problem. We can then concentrate on the hard cases (mistaken prediction or close to mistakes) and see what additional features would be useful for classifying them.

TABLE 1: List of codes that we will analyze, the numbers of nodes that each of the codes were run on, and a description of the algorithm that the code implements.

Code Name	# Nodes Used	Description of Algorithm Used
Cactus	64 and 256	Grid (Finite difference stencil for elliptic PDEs)
fvCAM-1d	64	Grid (Finite volume stencil)
fvCAM-2d	256	Grid (Finite volume stencil)
GTC	64 and 256	Particle/Grid (particle-in-cell)
GTC2	64 and 256	Particle/Grid (particle-in-cell)
GTC3	64 and 256	Particle/Grid (particle-in-cell)
Hypre	256	Sparse Linear Solver
LBMHD	64 and 256	Lattice (Boltzman complicated hexagonal stencil)
MHD2D	256	2D MHD equation solver
PARATEC	64 and 256	Fourier/Grid (plane wave DFT/FFT and BLAS3)
PMEMD	64 and 256	Particle (Particle-mesh Ewald similar to GTC)
SuperLU	64 and 256	Sparse Matrix Multiply

A list of the codes that we analyze is shown in Table 1.

4 Results

We have obtained results and applied a variety of machine learning algorithms,[1] including naïve Bayes, k-nearest-neighbor and the locally weighted learning algorithm.

In both cases, we reserve slightly over half of the test cases for training. Those include Cactus (64 nodes), fvCAM-1d (64 nodes), GTC (64 nodes), GTC2 (64 nodes), GTC3 (64 nodes), LBMHD (64 nodes), MHD2D (256 nodes), PARATEC (64 nodes) PMEMD (64 nodes), SuperLU (64 nodes).

[1]Implementations are from the *Weka* toolset [HFH+09].

As shown in Tables 2 and 3, the other test cases are used for testing. Note that GTC2 and GTC3 are not different codes, but are runs of the same code using different parameters.

In the first set of analyses, we apply four different machine learning methods to test cases in which each test case consists of one line per processor used in the run. Thus, the Cactus test case for 64 nodes is 64 lines long. Each line consists of seven fields, as follows:

1. the code used (e.g., Cactus),
2. the number of the node (e.g., 0–63),
3. the ratio of individual communication to collective communication on the node in question,
4. the number of other nodes that the node in question communicates with via point-to-point communication (e.g., `MPI_Send`) divided by the total number of nodes in the entire scheme,
5. the number of other nodes that the node in question communicates with via point-to-point communication (e.g., `MPI_Send`) multiplied by the number of bytes sent in that communication, divided by the total number of bytes sent in the entire scheme,
6. the total amount of collective communication initiated by the node,
7. the percentage of point-to-point communication initiated by the node that is asynchronous (e.g., `MPI_Isend` as opposed to `MPI_Send`)

Note that these fields merely represent a starting point. Further experimentation will be required to refine them.

TABLE 2: True positive rates for four different learning methods used for analyzing data for all processors. The test cases all use 256 nodes. (Numbers in parentheses represent cases where a GTC run is simply recognized as a different GTC run. Since it is the same code, we view this as a correct classification.)

Code Name	1NN	Naïve Bayes	Naïve Bayes Mult.	LWL
Cactus	0%	0%	0%	0%
fvCAM-2d	59.4%	97.5%	0%	0%
GTC	54.7% (100%)	0%	0%	1.6% (100%)
GTC2	1.9% (87.1%)	0%	0.4%	0.4% (100%)
GTC3	39.1% (100%)	0%	0%	98.4% (100%)
Hypre	0%	0%	0%	0%
PARATEC	0%	0%	99.6%	0%
PMEMD	8.2%	0%	0%	0%
SuperLU	13.1%	7%	0%	29.7%

The results of performing these analyses, for four different machine learning methods, k-nearest-neighbor, naïve Bayes, naïve Bayes multinomial, and locally weighted learning (LWL) are shown in Table 2. As shown, the results are not strong. The methods perform quite badly in most cases. Using k-nearest-neighbor, Cactus was misidentified as MHD2D, fvCAM-2d was mis-identified as Cactus, Hypre was mis-identified as GTC or MHD2D, and PARATEC, PMEMD, and SuperLU were all mis-identified as MHD2D. Using naïve Bayes, Cactus was mis-identified as MHD2D or PMEMD, all of the GTC algorithms and Hypre were mistakenly identified as SuperLU, and Paratec was mistakenly identified as PMEMD. Using naïve Bayes multinomial, all of the algorithms that were

misidentified were misidentified as MHD2D. Finally, using LWL, Cactus, fvCAM-2d, PARATEC, PMEMD, and SuperLU were misidentified as MHD2D, and Hypre was misidentified as GTC.

In the second set of analyses, we apply machine learning methods to test cases in which each test case consists of a single line. Each line consists only of thirteen fields, as follows:

1. the code used (e.g., Cactus),
2. the percentage of the total number of point-to-point calls in the program made by Node 0,
3. the percentage of the total number of point-to-point calls in the program made by Node 1,
4. the percentage of the total number of bytes in the program sent by Node 0 using point-to-point communication,
5. the percentage of the total number of bytes in the program sent by Node 1 using point-to-point communication,
6. the average ratio of individual communication to collective communication for node 0,
7. the average ratio of individual communication to collective communication for node 1,
8. the average number of other nodes that each node communicates with via point-to-point communication (e.g., `MPI_Send`), divided by the total number of nodes,
9. the number of other nodes that each node communicates with via point-to-point communication (e.g., `MPI_Send`) divided by $n \log n$ where n is the total number of nodes,
10. the number of other nodes that each node communicates with via point-to-point communication (e.g., `MPI_Send`) divided by the total number of nodes, squared,
11. the average number of other nodes that each node communicates with via point-to-point communication (e.g., `MPI_Send`) multiplied by the number of bytes sent in that communication, and divided by the total number of nodes,
12. the average amount of collective communication initiated per node,
13. the average amount of point-to-point communication initiated per node that is asynchronous (e.g., `MPI_Isend` as opposed to `MPI_Send`).

The results of performing these second analyses, using the same machine learning methods as discussed earlier in this paper, are shown in Table 3. These results are considerably improved in comparison to the first set of analyses. Note that in no case is Hypre classified correctly. This is not surprising, because it has no training data. Thus, if it were classified correctly, it would have had to be done either by virtue of process of elimination, or by chance. Thus, ignoring Hypre, using the k-nearest neighbor and naïve Bayes multinomial methods, the results show that eight out of nine algorithms were correctly classified.

In the codes that were not identified correctly, however, a similar algorithm was typically identified instead. For example, using the locally weighted learning method, fvCAM-2d was identified as Cactus another finite stencil algorithm.

TABLE 3: True positive rates for four different learning methods used for analyzing summary data, Node 0 data, and Node 1 data. The test cases all use 256 nodes. (Numbers in parentheses represent cases where an fvCAM or GTC run is simply recognized as a run of fvCAM or GTC, respectively. However, since it is the same code, we view this as a correct classification.)

Code Name	1NN	Naïve Bayes	Naïve Bayes Multi.	LWL
Cactus	100%	100%	100%	100%
fvCAM-2d	0% (100%)	0%	100%	100%
GTC	100%	0%	100%	0%
GTC2	100%	100%	0% (100%)	100%
GTC3	0% (100%)	0%	0% (100%)	0% (100%)
Hypre	0%	100%	0%	100%
LBMHD	100%	100%	0%	100%
PARATEC	100%	0%	100%	0%
PMEMD	100%	0%	100%	100%
SuperLU	0%	100%	100%	0%

5 Discussion

Why did these results turn out the way they did? Why is the analysis of nodes 0 and 1 so much more accurate than the analysis of the larger, and more complete amount of data? Our hypothesis is that in the analyses that we performed, the average statistics for the run were substantially more important than the per-node statistics. The reason is that the per-node statistics overweight the value of nodes other than node 0 and node 1, whereas node 0 is typically unique, acting as the leader for coordinating the operation of all other nodes. Further node 1 is largely representative of all other nodes. Thus, statistics for only two nodes need to be examined for the data to be understood, in most circumstances. While there is some variation between node 1 and nodes other than node 0, node 1 is *substantially* more similar to nodes other than node 0, than it is to node 0 itself. Thus, using data for more processors causes the data for nodes other than node 0 and node 1 to be overweighted, reducing the overall accuracy of an untuned version of the algorithm.

Fingerprinting HPC code—much like fingerprinting malware—would be relatively easy if the author or user of the code wanted the code to be fingerprinted. But the circumstances in which HPC code or malware runs are both potentially adversarial. There are always ways in which an attacker can take steps to defeat a set of defenses put in place. HPC systems are no exception. As with system and network-based IDSs, HPC code is also vulnerable to mimicry attacks [WS02]. That said, the value of an HPC system is in its ability to perform large computations quickly. Most codes are designed to perform as efficiently as possible. Any alterations to a particular code to mimic an authorized or safe algorithm would therefore likely slow down the code, perhaps substantially. Subtle variations in either performance on a single node, or between nodes can easily alter the performance of a particular piece of code by an order of magnitude, or more.

6 Conclusions and Future Work

The problems that we have outlined in this report have existed for years, and are non-trivial to solve. The ability to collect detailed MPI function call data at low performance overhead has enabled us to address the problem of fingerprinting HPC codes with a level of accuracy not previously possible. Thus, we have designed a technique that enables us to fingerprint HPC codes using runtime MPI data, with a high degree of accuracy. This work is preliminary, however, and there are many avenues

for future exploration of this work and related areas. For example, in this work, we selected and explored only a few metrics for comparison, but other metrics may very well exist that provide superior results, and countless other machine learning algorithms could be applied.

Separate techniques that we are investigating enable us to take this analysis further, with additional types of machine learning. For example, we have investigated machine learning and information theory to identify the most relevant features for classifying computations as normal or anomalous, while removing redundant features to improve generalization. A special type of Hidden Markov Model is inferred from communication patterns to concisely represent their statistical properties, and can be used to compute the likelihood that new communications are anomalous, as well as the distances between models of authorized computation.

Near-future work will determine if our approach can distinguish parallel computations whose communication patterns depend on varying inputs, such as the size or composition of the data sets that the program analyzes, or other initial conditions used. To address these concerns, we plan to incorporate additional data sources such as number of integer and floating point operations and cache miss rate of each processor. At the same time, after our initial educated guesses, will pursue a number of more structured approaches to identifying key features of the aggregated dataset. We will select the most informative of these data sources using metrics from information theory. In addition, we plan to construct several types of classifiers and combine them using a boosting technique, which improves classification accuracy by combining multiple, weaker classifiers. We will also design computations similar to those an attacker might use, in order to test the specific assumptions of our approach and evaluate its effectiveness. Ideally, we would also like to be able to fingerprint the code in a way that allows us to use "rules" or "signatures" rather than statistical variations, to increase accuracy. This is particularly important if this technique were deployed in an operational environment because false positives require human intervention and can take considerable time from system administrators to investigate.

We also we seek to fingerprint the codes in a way that enable us to not only distinguish between them, but to compare them in different ways. For example, we wish to compare codes on per-user basis to see when users are running code that looks like code they typically run or not [Lee09], or if they look like code that other users who have applied for an allocation on a supercomputer for similar purposes also use. We also wish to map the code that is run back to specific "dwarves," NAS Parallel Benchmarks (NPBs), or specific applications (e.g., supernova code, cryptography). We expect to analyze not only communication patterns as factors in our analysis, but also distinctions in the job submission process, and other system-level details captured in system and network logs. These additional categories of data will better help us to determine not only if a computer is being misused, whether by hackers, authorized users, or true insiders [BEP+08, BEP+09, BEF+10].

Ultimately, in order to validate these techniques, tens or even hundreds of sets of logs or traces of IPM data are needed, because data sets that are too small can cause classifiers to overtrain, and the resulting conclusions to be spurious [TM02]. In particular, we need different programs with the same number of nodes and the same inputs. Similarly, we need to examine the same program with different inputs, and different number of nodes assigned. Even the same code run more than once with exactly the same parameters, to evaluate the existence and/or effect of non-determinism are all essential elements to reaching some reasonable intellectual conclusion. Additionally, to map code to dwarves, one needs to gain a baseline of the dwarves to begin with, which we will do when reference implementations of the dwarves become available.

Finally, having identified much of the relevant data to collect, we will begin to use the results of these techniques to construct rules or signatures to identify allowed or disallowed behavior without having to rely on measuring statistical variations, and the pitfalls, such as false positives, that come with that technique.

7 Acknowledgements

This research was supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC02-05CH11231.

The author gratefully acknowledges and appreciates the interest and funding support of Kathy Yelick, helpful feedback and suggestions from Juan Meza, and the assistance with gathering and understanding the IPM data from David Skinner, Katie Antypas, Karl Fürlinger, Harvey Wasserman, and Nick Wright.

References

[ABC+06] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: a View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

[Axe00] Stefan Axelsson. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, Aug. 2000.

[Bac00] Rebecca Gurley Bace. *Intrusion Detection*. Macmillan Technical Publishing, 2000.

[BBB+91] David H. Bailey, Eric Barszcz, John Barton, David Browning, Russell Carter, Leonardo Dagum, Rod Fatoohi, Samuel Fineberg, Paul Frederickson, Thomas Lasinski, R. Schreiber, Horst Simon, V. Venkatakrishnan, and Sisira Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, March 1991.

[BCO+05] Julian Borrill, Jonathan Carter, Leonid Oliker, David Skinner, and Rupak Biswas. Integrated performance monitoring of a cosmology application on leading HEC platforms. In *Proceedings of the 34th International Conference on Parallel Processing (ICPP)*, pages 119–128, June 2005.

[BEF+10] Matt Bishop, Sophie Engle, Deborah A. Frincke, Carrie Gates, Frank L. Greitzer, Sean Peisert, and Sean Whalen. A Risk Management Approach to the 'Insider Threat'. In *Insider Threats in Cybersecurity*, Advances in Information Security Series. Springer Verlag, Berlin, September 2010.

[BEP+08] Matt Bishop, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. We Have Met the Enemy and He is Us. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW)*, Lake Tahoe, CA, September 22–25, 2008.

[BEP+09] Matt Bishop, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. Case Studies of an Insider Framework. In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS), Cyber Security and Information Intelligence Research Minitrack*, Waikoloa, HI, Jan. 5–8, 2009.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, Boston, MA, 2003.

[CDW04] Hao Chen, Drew Dean, and David Wagner. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, 2004.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[FB97] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997.

[FY02] Ahmad Faraj and Xin Yuan. Communication characteristics in the NAS parallel benchmarks. In *Proceedings of the Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 729–734, 2002.

[GT06] Carrie Gates and Carol Taylor. Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In *Proceedings of the 2006 New Security Paradigms Workshop (NSPW)*, pages 21–29, Dagstuhl, Germany, September 2006.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[HKB+09] Gilbert Hendry, Shoaib Kamil, Aleksandr Biberman, Johnnie Chan, Benjamin G. Lee, Marghoob Mohiyuddin, Ankit Jain, Keren Bergman, Luca P. Carloni, John Kubiatowicz, Leonid Oliker, and John Shalf. Analysis of Photonic Networks for a Chip Multiprocessor Using Scientific Applications. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 104–113, Washington, DC, USA, 2009. IEEE Computer Society.

[Hof99] Steven Hofmeyr. *An Immunological Model of Distributed Detection and its Application to Computer Security*. PhD thesis, University of New Mexico, 1999.

[JV91] Harold S. Javitz and Alfonso Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, 1991.

[KFL94] Calvin Ko, George Fink, and Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, pages 154–163, Dec. 5–9, 1994.

[KOPS09] Shoaib Kamil, Leonid Oliker, Ali Pinar, and John Shalf. Communication Requirements and Interconnect Optimization for High-End Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems (TDPS)*, 21(2):188–202, 2009.

[Lee09] Cynthia Bailey Lee. *On the User-Scheduler Relationship in High-Performance Computing*. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 2009.

[LMW+00] Barbara Staudt Lemer, Eric K. McCall, Alexander Wise, Aaron G. Cass, Leon J. Osterweil, and Stanley M. Sutton, Jr. Using Little-JIL to Coordinate Agents in Software Engineering. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 155–163, 2000.

[MHL94] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, May/June 1994.

[MTM+09] Chao Ma, Yong Meng Teo, Verdi March, Naixue Xiong, Ioana Romelia Pop, Yan Xiang He, and Simon See. An Approach for Matching Communication Patterns in Parallel Applications. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2009.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23):2435–2463, 1999.

[PBKM05] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Principles-Driven Forensic Analysis. In *Proceedings of the 2005 New Security Paradigms Workshop (NSPW)*, pages 85–93, Lake Arrowhead, CA, October 2005.

[PBKM07] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 4(2):137–150, April–June 2007.

[Pei07] Sean Philip Peisert. *A Model of Forensic Analysis Using Goal-Oriented Logging.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, March 2007.

[Rie06] Rolf Riesen. Communication Patterns. In *Proceedings of the Workshop on Communication Architecture for Clusters (CSC), Rhodes Island, Greece*, 2006.

[SKOS05] John Shalf, Shoaib Kamil, Leonid Oliker, and David Skinner. Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC)*, pages 17–30, 2005.

[SP10] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[TM02] Kymie M.C. Tan and Roy A. Maxion. "Why 6?" — Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188–201, Oakland, CA, 2002.

[VM03] James S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, 2003.

[WS02] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communication Security*, November 2002.

[ZHR+07] Jingmin Zhou, Mark Heckman, Brennan Reynolds, Adam Carlson, and Matt Bishop. Modelling Network Intrusion Detection Alerts for Correlation. *ACM Transactions on Information and System Security (TISSEC)*, 10(1), February 2007.