Position Paper: The Potential of Sampling for Dynamic Analysis

Joseph L. Greathouse Todd Austin

Advanced Computer Architecture Laboratory
University of Michigan
{jlgreath, austin}@umich.edu

Abstract

This paper presents an argument for distributing dynamic software analyses to large populations of users in order to locate bugs that cause security flaws. We review a collection of dynamic analysis systems and show that, despite a great deal of effort from the research community, their performance is still too low to allow their use in the field. We then show that there are effective sampling mechanisms for accelerating a wide range of powerful dynamic analyses. These mechanisms reduce the rate at which errors are observed by individual analyses, but this loss can be offset by the subsequent increase in test population. Nevertheless, there are unsolved issues in this domain that deserve attention if this technique is to be widely utilized.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—statistical methods; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, distributed debugging, testing tools

General Terms Performance, Security, Verification

Keywords Distributed Analysis, Sampling

1. Introduction

Dynamic software analyses observe programs as they run and make inferences about the correctness, performance, or security of any particular execution. Because they work alongside the executing program, these tests are able to observe situations that offline or static analyses have difficulty checking. Static tools, for example, may struggle to observe some errors due to the state space explosion problem caused by control-flow decisions. Dynamic analyses, in contrast, are able to test any path an analyzed execution takes.

Security tests such as taint analysis and correctness checks such as data race detection can help create robust software that is resilient to malicious attacks. However, they are unable to catch errors that lie on unexecuted paths. Consequently, dynamic analyses benefit from seeing large numbers of executions with a variety of inputs, allowing them to test more paths. Ideally, end-users would run these tests at all times, checking the plethora of potentially dangerous situations they encounter.

Table 1: Overheads from a selection of dynamic analyses

Analysis	Slowdown
Assertion Checking (Sec. 2.1)	5% – 2×
DIFT (Sec. 2.2)	3× – 150×
Race Detection (Sec. 2.3)	8× – 100×
Atomicity Checking (Sec. 2.4)	25× – 400×
Symbolic Execution (Sec. 2.5)	10× – 200×

Unfortunately, these systems suffer from very high runtime overheads. Table 1 shows some example dynamic analyses and the range of slowdowns they can cause. These large overheads present a twofold problem: they limit the number of test inputs that a developer can execute before shipping a piece of software, and they severely reduce the number of users willing to run the analyses. In both cases, high overheads hamper the tool's effectiveness.

Researchers have proposed solutions to help combat this problem. Zhao *et al.*, for instance, showed a number of mechanisms to accelerate shadow variables accesses [44]. Others have shown demand-driven analyses that are only enabled when operating on variables of interest to the analysis [16, 20]. There have also been works on parallelizing security checks [34] and decoupling the act of analysis from the original execution [8, 37].

These techniques do not completely solve the overhead problem. Umbra, for instance, reduced the overhead of shadow data accesses from 8–12× to 2–4× [44], but this is still more than most users would tolerate. This stymies the ability to distribute these tests to large populations of users.

In this paper, we make the argument that the best solution to this overhead problem is to sample the analysis space, allowing limited portions of the program to be analyzed during any individual execution. Ideally, the tool's ability to detect errors will be proportional to its overhead. This would allow users to control the overheads they experience while finding some percentage of the program's errors.

Because the entire dynamic execution will not be analyzed while sampling, no test is guaranteed to find all observable errors. We argue that distributing these sampled analyses across large end-user populations will offset the lower error detection rate. This larger population will also see many more inputs and dynamic states than a developer running a small test suite.

PLAS'11 June 5, 2011, San Jose, California, USA.

Allowing users to run these powerful analyses can increase the number of bugs caught and corrected before malefactors can take advantage of them. Dynamic analysis sampling makes this possible and therefore is a promising research direction. We believe the community should focus on solving the outstanding research and engineering problems that currently stand in the way of its widespread use.

2. Dynamic Security Analyses

Many dynamic software analyses can be used to reduce a program's vulnerability surface. Although they can guarantee neither correct execution nor the elimination of all bugs, they can still find many difficult-to-observe errors.

In this section, we review dynamic security analyses, their benefits, and their overheads. This list is not comprehensive, but an attempt to present a selection from the literature.

2.1 Assertion Checking

Some analysis systems begin their search for vulnerabilities by statically analyzing the code for operations that can be proven to be either secure or insecure. The tool can then insert dynamic checks around the remaining operations to ensure their safety at runtime. CCured used this combination of static and dynamic testing to secure programs against memory errors such as buffer overflows [33]. It is also possible to use this static/dynamic hybrid testing to perform checks such as ensuring type safety [11]. The overheads observed with these tools depends both of the complexity of the assertions and the power of the static analysis. The simple memory assertions inserted by CCured slow a selection of computationally-intensive applications up to 87% [10].

2.2 Dynamic Information Flow Tracking

Sometimes referred to as taint analysis, dynamic information flow tracking (DIFT) associates shadow values with memory locations and registers, propagates them alongside the execution of the program, and checks them to find errors. These shadow propagations form a dynamic dataflow that traces the movement of security-critical data throughout the program. Shadow variables can represent security information such as the trustworthiness or secrecy levels of data within the program, depending on the particular analysis.

One of the most commonly studied taint analyses is control flow attack detection [36, 41]. Other examples of this type of analysis include memory corruption detection [6], confidentiality verification [9, 29], malware analysis [43], and error tracing [3].

The performances of these systems can vary widely and depend on the type of analysis, the optimizations applied, the system used to construct the tool, and the inputs given to the particular execution of the program under test. Measured overheads for control flow attack detectors range from 3.6× [36] to upwards of 150× [20]. Other DIFT analyses see overheads of similar orders of magnitude: 20× [43] to hundreds of times [9] slower than native program execution.

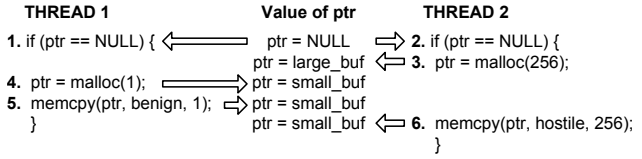


Figure 1: A data race resulting in a security flaw. The numbers represent dynamic execution order. Both threads attempt to initialize the shared pointer (1, 2). Due to the lack of synchronization, thread 1 allocates a small buffer (4), but thread 2 then copies a large amount of data into it (6). This particular ordering can cause a buffer overflow, exposing a potential security flaw.

2.3 Data Race Detection

Dynamic data race detectors calculate if it is possible for two threads that access a shared memory location to do so in an unordered way. These unordered accesses can lead to nondeterministic operation or crashes when programs access variables in unintuitive, undesired orders. While this may not seem to be a security problem at first, Figure 1 shows an example of a data race that creates a memory corruption vulnerability. This example is drawn from a security flaw found in some versions of the OpenSSL TLS library [32].

A number of commercial and open source race detectors exist [4, 35, 38, 42]. Their performance depends heavily on the tool and its optimizations, as well as the amount of sharing seen within any particular dynamic execution of the program. Our tests put current versions of Helgrind at about a 100× slowdown, while Serebryany and Iskhodzhanov list slowdowns of between 20× and 50× for Google ThreadSanitizer [38]. FastTrack is an example of work on lowering the overheads of a dynamic data race detector; it is able to reduce slowdowns to about 8.5× for some benchmarks [13].

2.4 Atomicity Violation Detection

Atomicity violations are concurrency errors that are similar to, yet distinct from, data races. In parallel code, a programmer may desire that some collection of variables be accessed atomically (*i.e.* no other thread accesses any of the variables until the first thread finishes working on them). However, even if variables are locked in a way that prevents data races, this atomicity is not guaranteed [12].

Lu *et al.* looked at a number of concurrency bugs in large open-source programs and found that atomicity violations were quite common [26]. Figure 2 demonstrates how these flaws, much like data races, could result in security errors. Even though the accesses to the shared pointer have locks surrounding them, meaning that race detection tools would not indicate an error, the threads can still be ordered in such a way that the buffer overflow exists.

Atomicity violations are difficult to automatically detect, as they are a disconnect between what the programmer *wants* and what she *commands*. While the designer may assume that two regions are executed atomically, a tool cannot learn this assumption from the code. As such, current tools that find these errors focus on training over many known-good

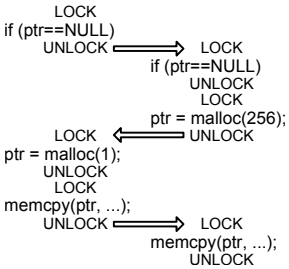


Figure 2: Atomicity violation security flaw. This version of the code has locks around every access to the shared pointer. A race detector will not see this as an error. Nonetheless, it is still possible to order the threads such that the buffer overflow exists.

executions, recording the observed atomic regions, and later verifying that these regions are always accessed atomically. The online detection mechanisms for these tools have slowdowns that range from 25× [27] to upwards of 400× [31].

2.5 Dynamic Symbolic Execution

Symbolic execution follows a program as it runs and, rather than working with concrete values for any particular variable, attempts to calculate what *possible* values could exist in the program. As the program calls functions with known return values, passes conditional statements, and uses concrete values, the limits on any particular variable can be constrained. These symbolic values can be checked to find inputs that could cause errors such as memory corruption [23].

The system by Larson and Austin looks only at the control path executed on any particular run, constraining symbolic variables to match the bounds that are imposed by the current path. It tests memory accesses along the executed path to find if they could be corrupted by slightly different inputs that still made it to this state. Tools like DART, on the other hand, start with completely unconstrained inputs and systematically search for bugs in all possible control paths [14]. Godefroid *et al.* claim that their symbolic execution is "many times slower than [running] ... a program", while Larson and Austin show overheads between 13× and 220×.

3. Sampling for Dynamic Analyses

Because they can only test a single path through a program per execution, dynamic analyses benefit from observing numerous program executions with different inputs. Figure 3 demonstrates a simple example of this. If an error existed in basic block D of this control flow graph, a dynamic analysis would be unable to observe it until the program executed the code within that block. In this case, it would take an average of 10,000 executions of the program before the tool would observe the problem. Note that we are ignoring (for the sake of brevity) that a static analysis would find this error easily; this limitation is also true in more complicated examples.

To this end, dynamic analyses can be more effective when performed by a large population of tester. Ideally, these tests would be run by end-users, as they run the program more often than developers and use inputs that developers may not think to test. Unfortunately, it is unlikely that end-users would be willing to purchase programs that have been modified to be 2×, let alone 200×, slower. Few would purchase a faster computer to run security checks, and it is doubtful that users who pay for CPU time (such as cloud computing customers) would wish to centuple their costs in order to find errors, regardless of the security benefits.

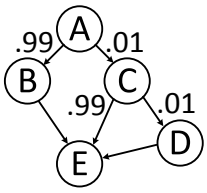


Figure 3: Dynamic analysis may not always catch bugs. This is a control flow diagram with weighted random probabilities at each branch. It takes an average of 10,000 executions to run the code within basic block **D**. If an bug exists there, a dynamic analysis will only find it 0.01% of the time.

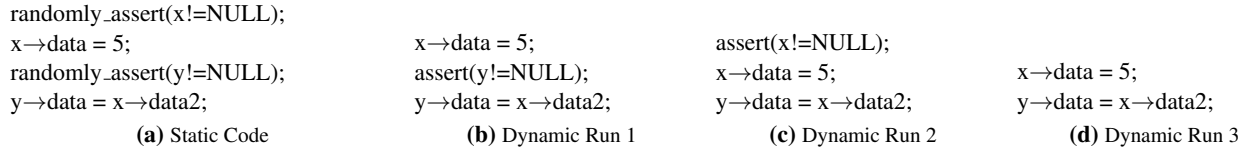
The following sections look at works that have increased the performance of a variety of dynamic analyses by performing *sampling*. In such systems, small portions of the program are analyzed during each run, allowing performance increases over a system that constantly checks the program. In a well-designed sampling system, the ability to observe errors is related to the sampling rate and performance. Lower sampling rates increase the performance but lower the probability of finding any particular error.

This accuracy reduction can potentially be offset by the large increase in the number of testers. As a simple example: if sampling analysis catches 10% of the executed errors in a program, and were used to test the example in Figure 3, it would catch the error once in every 100,000 executions. If 5000 users were to run these analyses, each would only need to run the program an average of 20 (rather than 10,000 or 100,000) times before the error would be observed somewhere and a report could be returned to the developer. This increase in population not only offsets the accuracy loss due to sampling, but alleviates some of the problem of only observing errors on executed paths.

This paper will not go into details of the mechanisms for bug triage with this type of system. However, Liblit and others have done a great deal of work designing mechanisms for pinpointing the cause of errors using the numerous reports sent from highly distributed bug detection systems [7, 25].

3.1 Sampling for Performance Analysis

While performance analysis is not a security test, it is educational to review the works on sampling in this area. Arnold and Ryder showed software mechanisms for sampling performance analyses [1], as did Hirzel and Chilimbi [19]. These mechanisms, in a broad sense, enter the analysis on certain software conditions, profile some instructions, and then leave analysis until the next condition. Hirzel



(a) Static Code **(b)** Dynamic Run 1 **(c)** Dynamic Run 2 **(d)** Dynamic Run 3

Figure 4: Sampling for assertion analyses. This system has a random probability of performing any particular test. It is possible that `x==NULL` in, e.g., (4b). However, with a large population, error will be caught proportional to the percent of time each assertion is enabled.

and Chilimbi showed that they were able to gather accurate performance profiles at overheads of overheads as low as 3–18% (compared to 30%–10× for non-sampled analysis).

The concept of sampling is so well ingrained into the performance analysis community that commodity processors have dedicated performance sampling hardware. While taking interrupts on performance counter rollover is one of the most common ways of analyzing samples of events [40], modern processors include sampling facilities such as Intel's Precise Event Based Sampling (PEBS) [21]. Such mechanisms amortize interrupt costs by automatically storing information on a subset of instructions or events into physical memory buffers without involving software.

3.2 Sampling for Assertion Checking

Liblit *et al.* described an instruction-based method of sampling assertion tests, as demonstrated in Figure 4 [24]. As long as the checks are not related to one another, it is possible to perform sampling by only enabling a random subset of the assertions during each execution. They showed that performing this type of sampling was able to lower the maximum overhead of their analysis from 2.81× to 26% when sampling one out of every thousand checks. They demonstrate that such sampling rates can find uncommon errors with user populations of moderate size.

Hauswirth and Chilimbi performed similar types of sampling in order to do memory leak detection [18]. Their sampling mechanism enables checks for groups of instructions so as to reduce the overhead of deciding when to enable the tool. They also showed that sampling reduces overheads (from an estimated 5× to 5%), though about 7% of their error reports falsely identified active pages as leaked.

3.3 Sampling for Dynamic Information Flow Tracking

We recently described mechanisms for sampling dynamic dataflow analyses, such as those described in Section 2.2 [15]. Because false positives and extraneous false negatives can occur if we randomly sample code, dataflow analysis sampling requires that instructions operating on related data be analyzed together. It is possible that these instructions are not located near to one another (in space or in time), so we described the concept of data-based sampling, as shown in Figure 5. Rather than observing a subset of instructions, data-based sampling attempts to analyze a subset of the program's dataflows. We previously described such sampling in a hardware-assisted tool, though that system could not directly choose to stop analysis to lower overheads [17].

Arnold *et al.* showed a similar mechanism of object-based analysis sampling that used the extensive program information available to a Java virtual machine. Instead of checking a subset of dynamic tests, their system only performed analyses (ranging from assertion checks to techniques such as typestate checking) on a subset of instantiated objects [2].

Both works demonstrated that this type of sampling gives users control of performance, with the lowest overheads sitting within a few percent of a system running no analysis. Nonetheless, they maintained high error detection accuracy relative to the performance. We showed that one particular error could be found 0.1% of the time at a 10% slowdown.

3.4 Sampling for Concurrency Tests

LiteRace logs memory accesses and synchronization operations and performs offline data race detection on them. As Marino *et al.* point out, logging a small, random sample of the memory accesses will allow many races to be detected while significantly reducing the logging overhead. All synchronization points must still be logged, or the sampling system may cause false positives. By choosing to log cold-path code at a much higher rate, they were able to reduce the runtime overhead from 651% to 28%, while still detecting about 70% of the observable data races [28].

PACER performs online race detection, but enables the detector for long intervals. When the analysis is later disabled, each variable is sent through one last check when it is next accessed. In doing so, Bond *et al.* showed that it is possible to find races in rough proportion to the sampling rate while still allowing control over performance [5].

There is little work on sampling atomicity violation detection in the classic sense. Rather than attempting to sample previous atomicity violation mechanisms, Jin *et al.* look at a new way of detecting atomicity violations by keeping track of samples of data sharing statistics from many user runs. They then trace program crashes back to these errors using statistical analyses, finding both data races and atomicity violations [22]. This type of analysis would not be possible without sampling, showing that *distributed sampling can enable new analyses that were not possible before*.

4. Future Research Directions

The previous works on dynamic analysis sampling demonstrate the power and effectiveness of such techniques. In these tools, sampling has allowed users or developers to control the slowdowns caused by dynamic correctness checks,

(a) Static Code **(b)** Dynamic Run 1 **(c)** Dynamic Run 2 **(d)** Dynamic Run 3

Figure 5: Sampling for DIFT. Instead of enabling analysis for random individual instructions, DIFT sampling must attempt to follow entire shadow dataflows. If a point early in the dataflow is skipped, all values later in the dataflow could be incorrect, leading to needless false negatives. Care must also be taken in the sampling system to avoid false positives.

significantly reducing overheads at low sampling rates. Despite these benefits, there are still unsolved research questions in this area. Examples include:

- There are analyses that currently have no sampling mechanisms. Examples include more complicated DIFT tests, symbolic execution, and traditional atomicity violation detection systems. It may not be trivial to sample these. For example, naïve symbolic execution sampling may set symbolic variables to concrete values when analysis is forcibly disabled. This may cause the tool to lose much of its power if analysis is disabled often.
- There may be new dynamic analyses that are possible only in distributed sampling systems. The cooperative concurrency error system described by Jin *et al.* is an example of this [22]. The mechanisms they describe cannot directly detect bugs and would have high overheads in non-sampling systems. Nevertheless, by analyzing large numbers of these observations sampled from buggy runs, it is possible to find the locations of concurrency errors at low overhead. Can sampling be used to make new tests?
- There may be better ways of performing data-based sampling. Current techniques randomly remove meta-data with no regard for the difficulty of analyzing very large dataflows. The probability of completely observing a large dataflow is therefore described by a geometric distribution on the number of attempts to stop. There may be better ways of doing this, or perhaps mechanisms could be built to optimize the choices over multiple executions.
- Similarly, it may be possible for the distributed analysis systems to coordinate their sampling decisions. In this case, it could be possible to optimize these decisions to attain high coverage for the whole population. These choices will probably depend on the type of analysis.
- It would be interested to study the performance decreases and latency increases that most users would tolerate in sampling systems. Few studies focus on such topics; our searches mostly found decades-old work with such claims as "...response should begin within two second..." [30]. Works such as that by Shye *et al.*, that look directly at user dissatisfaction with slowdowns, are an excellent step in this direction [39].

There are also engineering problems that must be solved if analysis sampling is to be used in practice.

- Few dynamic analysis tools currently allow sampling. No Valgrind tools, commercial race detectors, or other systems of this nature allow users to sample their analyses. High-quality analysis systems would need to offer a push-button option to enable sampling before most developers could to utilize this technology. Liblit's Cooperative Bug Isolation project allows some simple instrumentation sampling [25], as does Holmes, which integrates into Microsoft Visual Studio [7]. However, both are primarily used to link crashes to their most likely source, rather than probabilistically find errors as they occur.
- There are few libraries to help sampling and distributed bug reporting in dynamic analysis tools. Tool writers who use Valgrind, for instance, do not have ready-made sampling libraries; they would instead need to rewrite these mechanisms for each tool. Tool writers would also need to begin integrating libraries such as Google's Breakpad into their tools if they expect to return bug reports.
- Limited back-end structure for accepting distributed reports exists. Even if developers were to modify their tools to perform sampling, the mechanisms for accepting these reports, performing bug triage, and statistically inferring which bugs lead to errors are not well-supported within the dynamic analysis community. Most tools currently emit error reports to local files or to standard outputs, as they are designed to be run only by the developer. There would need to be work to integrate systems such as Mozilla's Socorro into any popular software that began to use distributed software analysis.

In summary, while dynamic analysis sampling is an area of work that promises highly-beneficial results, it also has a number of interesting research and engineering topics remaining. We feel that dynamic analysis sampling has great future potential and should be the focus of further work within the dynamic security analysis community.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, 2001.

[2] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, 2008.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[4] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling data race detection in the Intel Thread Checker. In *Workshop on Software Tools for MultiCore Systems*, 2006.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *PLDI*, 2010.

[6] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN*, 2005.

[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[8] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[10] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI*, 2003.

[11] C. Flannagan. Hybrid type checking. In *POPL*, 2006.

[12] C. Flannagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[13] C. Flannagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *PLDI*, 2009.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[15] J. L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco. Highly scalable distributed dataflow analysis. In *CGO*, 2011.

[16] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *ISCA*, 2011.

[17] J. L. Greathouse, I. Wagner, D. A. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie. Testudo: Heavyweight security analysis via statistical sampling. In *MICRO*, 2008.

[18] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, 2005.

[19] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *FDDO*, 2001.

[20] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys*, 2006.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*.

[22] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[23] E. Larson and T. Austin. High coverage detection of input-related security faults. In *USENIX Security*, 2003.

[24] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[25] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

[26] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[27] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[28] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[29] S. McCamant and M. D. Ernst. A simulation-based proof technique for dynamic information flow. In *PLAS*, 2007.

[30] R. B. Miller. Response time in man-computer conversational transactions. In *AFIPS Fall Joint Computer Conference*, 1968.

[31] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In *MICRO*, 2010.

[32] National Vulnerability Database. Vulnerability Summary for CVE-2010-3864: OpenSSL 1.0.0. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3864`, 2010.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM TOPLAS*, 27(3):477–526, 2005.

[34] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[35] Y. Qi, R. Das, Z. D. Luo, and M. Trotter. MulticoreSDK: A practical and efficient data race detector for real-world applications. In *PADTAD*, 2009.

[36] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, 2006.

[37] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools. In *PLDI*, 2010.

[38] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[39] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. A. Dinda, and R. P. Dick. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *MICRO*, 2008.

[40] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[41] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[42] C. Terboven. Comparing Intel Thread Checker and Sun Thread Analyzer. *Parallel Computing: Architectures, Algorithms and Applications*, 38:669–676, 2007.

[43] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.

[44] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *CGO*, 2010.