Highly Scalable Distributed Dataflow Analysis

Joseph L. Greathouse, Chelsea LeBlanc, Todd Austin and Valeria Bertacco
Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor
{*jlgreath, leblancc, austin, valeria*}*@umich.edu*

Abstract

Dynamic dataflow analyses find software errors by tracking meta-values associated with a program's runtime data. Despite their benefits, the orders-of-magnitude slowdowns that accompany these systems limit their use to the development stage; few users would tolerate such overheads.

This work extends dynamic dataflow analyses with a novel sampling system which ensures that runtime slowdowns do not exceed a user-defined threshold. While previous sampling methods are inadequate for dataflow analyses, our technique efficiently reduces the number and size of analyzed dataflows. In doing so, it allows individual users to test large, stochastically chosen sets of a process's dataflows. Large populations can therefore, in aggregate, analyze a larger portion of the program than is possible by any single user running a complete, but slow, analysis. In our experimental evaluation, we show that 1 out of every 10 users expose a number of security exploits while only experiencing a 10% performance slowdown, in contrast with the 100× overhead caused by a complete analysis that exposes the same problems.

1. Introduction

Dynamic software analysis is a powerful mechanism for exposing software errors and inefficiencies. Tools in this domain monitor the runtime state of a program and observe situations that may only arise during actual execution. Performance profilers, for instance, can pinpoint software inefficiencies by observing running processes. Dynamic analysis tools that detect software errors instead look for violations of runtime properties.

Dynamic dataflow analyses associate meta-data, also called shadow data, with memory locations and then propagate and check these values in conjunction with the program's original operation. Taint analysis follows "untrusted" data through a process to determine whether it is used without being properly validated [26]. Similarly, dynamic heap bounds checkers such as Valgrind's Annelid tool associate pointers with heap allocations and verify that every dereferenced value points to a location within the valid range of its region [23].

The power of these dynamic analysis tools comes with costs: only the portions of an application observed during a particular execution can be analyzed, and the performance overheads introduced by these approaches make it difficult to observe many different executions. Some Valgrind analyses slow execution by over 1,000 times [16]. No end-user would be willing to run such analyses, and developers performing these tests often use simplified inputs to reduce runtimes. This leads to myopic analyses and reduces the effectiveness of the testing.

This paper presents a novel software-based technique that limits the per-execution cost of dataflow analyses by distributing them across multiple program runs and/or over many users. In this system, a software overhead manager observes the runtime impact of the analysis procedures and can constrain the analysis to a set of stochastically chosen shadow dataflows once system performance degrades below a user-specified threshold. This allows users to cap the slowdown they experience, removing one of the main limitations of dataflow analysis. Moreover, developers can run larger, more insightful tests, and large end-user populations can run analyses that were formerly restricted to a development setting. Beta testers, production servers, and even mainstream users can then analyze programs in the background, enabling unprecedented levels of high-quality software analysis.

We built a prototype of this distributed dataflow analysis system by modifying the Xen hypervisor [4] to work with an augmented version of the emulator QEMU [5]. We implemented a taint analysis system within this prototype and performed experiments that demonstrate that the system provides fine-grained control of overheads while still delivering high-quality and highly scalable analyses. Experiments with real-world security exploits show that our solution easily exposed the security flaws in our test applications with a small population of users while significantly reducing the performance overhead that individual users experienced.

This work makes the following contributions:

- We show that **previous code-based dynamic sampling systems are inadequate for dynamic dataflow analyses**. They sample programs' instructions and thus do not take into account the effects of shadow

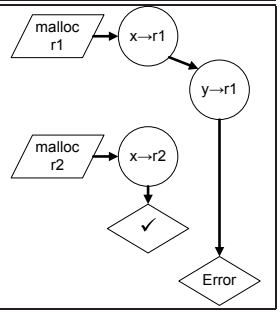
Dynamic Instructions	Analysis Operations	Data Values	Shadow Values	Dataflow
pointer x = malloc(10);	create r1 x.region ⇐ r1	x: a	r1.start: a, r1.end: a+9 x.region: r1	
pointer y = x + 15;	y.region ⇐ x.region	x: a y: a+15	r1.start: a, r1.end: a+9 x.region: r1, y.region: r1	
x = malloc(20);	create r2 x.region ⇐ r2	x: b y: a+15	r1.start: a, r1.end: a+9 r2.start: b, r2.end: b+19 x.region: r2, y.region: r1	
dereference(x);	x.region.start ≤ x ≤ x.region.end ? **Yes. No Error.**	”	”	
dereference(y);	y.region.start ≤ y ≤ y.region.end ? **No. Boundary Error.**	”	”	

Figure 1: Example Dataflow Analysis. The schematic illustrates how a dynamic bounds checker associates allocated memory regions with pointers and, by propagating the association to any derived pointers, builds a dataflow that connects pointers to heap regions. By checking a dereferenced pointer's address against the bounds of its region, it is possible to find heap errors such as overflows and uses-after-free.

dataflows when disabling an analysis. This leads to both false positives and a large number of false negatives.

- We present a new technique for performing sampling in dynamic dataflow analysis systems. By sampling dataflows, rather than instructions, we are able to **effectively control analysis overheads** while avoiding many of the inaccuracies of previous sampling systems.
- By allowing individual users to control the sampling rate, we **enable the distribution of dynamic analyses to large populations**. We show that our sampling system is able to observe a large fraction of the errors that a traditional analysis can discover but at much lower performance overhead.

This paper is organized as follows: we review background works in Section 2. We then present a method of dataflow-oriented sampling in Section 3 and detail our implementation in Section 4. Section 5 presents our experimental evaluation, demonstrating that we can provide strong analyses while controlling performance overheads. Finally, we discuss other related works in Section 6 and conclude in Section 7.

2. Background

In this section, we present background concepts that we leverage in the presentation of our solution. Because our technique is designed to accelerate dynamic dataflow analyses, we first summarize how these systems operate. We then look at demand analysis, a technique that offers, but does not guarantee, lower performance overhead when performing dynamic dataflow analyses. Finally, we discuss previous works on sampling dynamic analyses and show how they are inadequate for dataflow sampling.

2.1. Dynamic Dataflow Analysis

Figure 1 shows an example of dynamic dataflow analysis as performed by a heap bounds checker, a system that associates heap meta-data with pointers. The meta-data details the heap objects to which each pointer refers, and it is moved throughout the program along with the pointers and their derived values. When a pointer is dereferenced, the checker verifies that it correctly points within its associated object.

In the figure, parallelograms represent points that allocate memory from the heap. These calls to malloc normally only return pointers to the allocated regions. In this heap analysis system, they also produce meta-data that describe the regions and associations between pointers and these regions (represented by circles that list their region associations). As pointers are copied, the shadow values associating pointers with regions are also copied into the destination variables; this flow of shadow data is represented by arrows between two circles. Finally, diamonds represent the validity checks that take place when a pointer is dereferenced. A checkmark means that a pointer correctly addressed its associated region, while an error means it did not.

In the example, a 10-byte memory region, *r1*, is first allocated from the heap, and its start address, a, is stored into the pointer *x*. The shadow values of this memory region list this start address and the end address of the region, $a+9$. The pointer *x* also has a shadow value indicating that *x* is associated with *r1*. The pointer *x* is next used in an arithmetic operation that stores the value $a+15$ into another pointer, *y*. Because *x* was used as the source value for the operation, the shadow value associated with *y* receives the same region association as *x*.

The pointer *x* is then changed to point to a newly associated 20-byte memory region, *r2*. This region begins at address b and has its own meta-data storing its start and end addresses. This also means that the shadow value for *x* now associates it with *r2*.

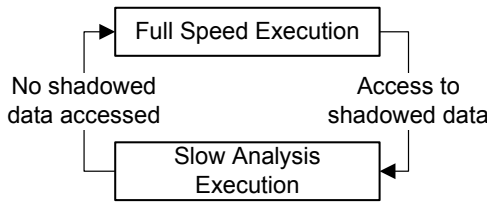


Figure 2: Traditional Demand Analysis. This technique attempts to execute the program normally and enables slow analysis only when shadowed data is encountered.

When the next instruction dereferences x, the analysis system checks that the value of x, b, is between b and $b + 19$, the start and end addresses of *r2*. This address is within the region, so the dereference is considered safe. The last instruction attempts to dereferences y, leading the analysis system to compare the value of y, $a + 15$, to the limits of its associated region, *r1*. Because the end of *r1* is at $a + 9$, the dereference is not safe and an error is raised.

Heap bounds checking can, as shown, be used to verify that accesses to memory regions are correct, a commonly desired security and correctness feature in type-unsafe languages [23]. A number of other powerful tools also use dynamic dataflow analyses. Taint analysis, for instance, marks memory locations as untrusted when their values originate from unsafe sources (*e.g.* a network buffer). By propagating this information to every value derived from this untrusted data, it is possible to detect a number of common security attacks [26]. Valgrind's Memcheck tool keeps track of whether a value has been initialized (or is derived from an initialized location), and can find memory accesses that could potentially cause crashes [25]. Dynamic race detectors, such as Intel's ThreadChecker, associate vector clocks with each memory location and perform "happens-before" calculations on each access to detect potential races among multiple threads [3].

The power of dynamic dataflow analyses can be undermined by the overheads they introduce. Simple systems such as Memcheck result in slowdowns on the order of 20× [25], while more complicated tools such as taint analyses can result in overheads beyond 100× [14]. Though the dataflows in a race detector are simpler (the meta-data is only propagated from synchronization points), the analysis performed on each memory access is much more complex: these systems can introduce worst-case overheads as high as 1,000× [16]. These extreme overheads present a twofold problem: they limit adoption, but more insidiously, they significantly reduce the number of dynamic situations that can be observed within a reasonable amount of time. Because dynamic analysis tools benefit from observing multiple and varied runtime situations, performance overheads have a strong impact on their ability to find errors.

2.2. Demand-driven Dataflow Analysis

Demand analysis is a method to mitigate these performance issues. In this technique, software that is not manipulating meta-data is not analyzed and thus runs much faster. Figure 2 illustrates this type of system. The software begins by executing natively, with no analyses taking place. However, when it encounters shadowed data[1], the process switches to an alternate mode of execution which also performs the complete (but slow) dataflow analysis. If the analysis system later finds that it is no longer operating on shadowed data, the process is transitioned back to executing with no analysis. Demand analysis leads to higher performance when a program rarely operates on shadowed data.

Unfortunately, demand analysis cannot provide performance guarantees. Processes may continually operate within the slow analysis tool if they frequently access shadowed data, yielding no performance improvement. This can be the case for certain worst-case inputs, as Ho, *et al.* demonstrated for their demand taint analysis system [14], or for tools in which nearly all data is shadowed, such as bounds-checkers.

2.3. Code-based Sampling and its Deficiencies

Sampling is a popular method of reducing dynamic analysis overheads. In general, a sampling system attempts to limit the amount of time that an analysis is enabled while still building an accurate picture of the program's characteristics despite the lack of a complete analysis. Arnold and Ryder describe a system where a program runs analysis code only during a portion of its execution and show that it is an accurate method for gathering runtime profiles of a program while suffering little slowdown [1].

These code-based sampling systems analyze a percentage of the instructions within a program's dynamic execution; the probability of performing analysis on any particular instruction is the only parameter available to control performance overhead. Figure 3a illustrates one negative effect this may have on dynamic dataflow analysis systems: this example uses the same bounds checking analysis shown in Figure 1, except that the final check of the variable y is skipped when the sampling system attempts to reduce the performance overhead. This results in false negatives, an effect inherent to any sampling system.

While code-based sampling is effective for some dynamic testing techniques, it is both inefficient and incorrect for dataflow analyses. By skipping the analysis of dynamic instructions without concern for their shadow values, it

1. We refer to meta-data itself as *shadow data*. Variables that have associated meta-data are called *shadow**ed** data*.

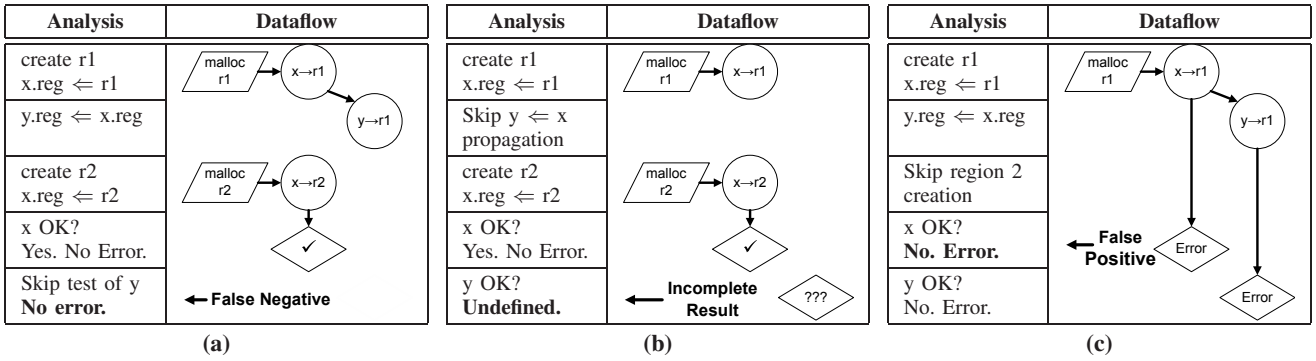


Figure 3: Code-based Sampling Fails for Dataflow Analysis. (a) Skipping checks may lead to false negatives. **(b)** Sampling instructions within a dataflow analysis may cause false negatives later in the program due to dataflow changes. **(c)** Worse still, instruction sampling may modify a dataflow in ways that cause false positives.

is possible (in fact, likely) that the shadow dataflows will differ from the dataflows of the associated variables. These incorrect shadow dataflows may cause the system to catch fewer errors, such as when shadow values indicating erroneous states are never written. This is illustrated in Figure 3b, where skipping an earlier shadow propagation results in the system missing the boundary error later in the program. This problem will compound if the dataflow leading to an error goes through multiple instructions and the probability of analyzing any individual instruction is not high.

An even more worrisome case occurs when an out-of-date shadow value leads to false positives (*i.e.*, reporting errors that do not exist). Figure 3c demonstrates this scenario. If analysis is disabled during an operation that should change a variable's region association, the corresponding meta-data remains unchanged. When analysis is later re-enabled, the bounds checker reports an error because the actual pointer addresses a different memory region than the meta-data. **Code-based sampling techniques often skip meta-data manipulations because they selectively enable instrumentation on instructions. This leads to dangerous false positive and undesirable false negatives.** A sampling system for dataflow analyses must therefore always be aware of the shadow dataflows on which it operates.

3. Effectively Sampling Dataflow Analyses

To remedy the deficiencies of code-based sampling techniques, we present a method for sampling dataflows instead of instructions. Rather than disabling analysis for instructions that may modify shadowed data, we instead skip the analysis of entire shadow dataflows.

As a program runs, a dynamic analysis system can create numerous shadow dataflows. A deterministic program that is run multiple times with the same inputs will see the same outputs and the same shadow dataflows. Similarly, a program run multiple times with different inputs may have dataflows that repeat. In these cases, we do not need to analyze all dataflows every time to find an error. If we analyze a dataflow that leads to an error, we can report the problem and skip the analysis of this dataflow in future runs.

Because we cannot know ahead of time where errors reside (knowing this would make analysis superfluous), we cannot decide ahead of time to skip dataflows that do not lead to errors. A scalable distributed system should not require all nodes to communicate with one another, so we also make the assumption that multiple users communicate neither with one another, nor across multiple executions. This means that our sampling system cannot coordinate between executions to determine which dataflows to analyze. It instead randomly selects some dataflows and ignores all others. During the next execution (or for the next user), a different set of dataflows will be analyzed. If the dataflows chosen for analysis are appropriately random, then enough users will, in aggregate, find all observable errors.

The benefit of only observing a subset of the total dataflows is that the analysis system causes less overhead as fewer dataflows are tested. In fact, we can take advantage of this to allow users to control the analysis overheads they experience. By setting the number of observed dataflows, the user can indirectly control slowdowns. Similarly, the system can directly cap performance losses by removing dataflows after crossing a performance threshold. The instructions that operate on these removed dataflows can subsequently execute without requiring slow analysis.

Figure 4 illustrates the operation of our dataflow sampling system. Like a demand analysis system, it disables analysis when it is not operating on shadowed data and activates the analysis infrastructure whenever such data are encountered. However, if the tool's performance overhead crosses a user-defined threshold, it begins to probabilis-

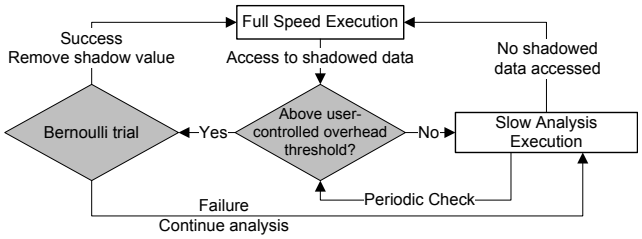


Figure 4: Dynamic Dataflow Sampling. Our system samples a program's dataflow by stochastically removing shadow values after passing a performance threshold. When combined with demand analysis, this enables dataflow, rather than instruction, sampling.

tically discard the shadow dataflows that it encounters. In a taint analysis system, for instance, it will randomly untaint some untrusted variables, implicitly marking them as trusted. The removed meta-data should be chosen in some stochastic manner in order to guarantee that different dataflows are observed during each program execution. We model this as a Bernoulli trial.

At this point, the demand analysis system allows native execution to resume, as it is no longer operating on shadowed variables. Eventually, as the program encounters fewer shadow values, it will spend less time in analysis and will begin executing more efficiently. The observed slowdown will eventually go below a threshold, and the sampling system can stop removing shadow variables. This may lead to further slowdowns, restarting the stochastic meta-data removal process. Alternately, the system may simply continue to operating in a demand analysis fashion.

As with other sampling approaches, these performance improvements come at the cost of potentially missing some dynamic events. However, when we remove shadow dataflows (rather than instrumentation associated with instructions), we not only reduce the amount of slowdown the user will experience in the future, but we also maintain the integrity of the remaining dataflows, thereby preventing false positives. As the example in Figure 5 shows, eliminating meta-data removes an entire dataflow, which leads to reduced overheads. The remaining meta-data is still propagated and checked as before, thereby eliminating the false positives that were possible with the code-based sampling of Figure 3c. **Our dataflow sampling technique analyzes samples of dynamic dataflows, rather than dynamic instructions. In doing so, it yields performance improvements without introducing false positives.**

Besides false positives, sampling systems must deal with false negatives (*i.e.* missing a real error). Although all sampling systems result in some missed errors, an increased population of testers can help offset this. Because individual users experience much lower overheads, the total population of users willing to run the tests can increase. This may

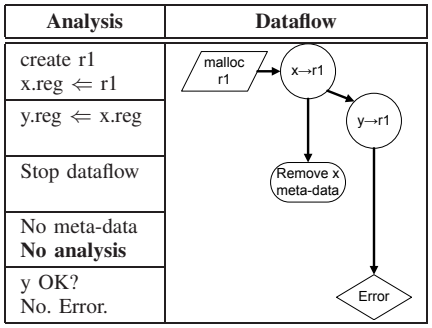


Figure 5: Dataflow Sampling Prevents False Positives. Eliminating a shadow value removes the entire dataflow that depends on it. No false positives occur because all propagations and checks related to that data are skipped.

offset the errors missed due to the false negatives caused by sampling, though we do not further speculate about the number of extra users our technique would deliver. We instead focus on keeping our false negative rate reasonably low.

The method of choosing which shadow values to remove should strive to observe many different dataflows across each execution so that numerous uncoordinated users can analyze varied parts of the program. Ideally, each execution should observe as many dataflows as possible, though this may lead to overly high slowdowns that cause the sampling system to engage. Because small dataflows are likely to be completely observed by many of the numerous disparate analysis systems, it may be beneficial to prioritize the more-difficult-to-analyze large dataflows. In this paper, we choose the discarded shadow values in a uniform random manner, and leave more complicated methods for future work.

4. Prototype System Implementation

We built a prototype of our dataflow sampling system on top of a demand analysis system that uses the Xen virtual machine monitor [4] and the emulator QEMU [5]. Entire virtual machines, called domU domains in Xen, are monitored in this design. Data that enter the virtual machine from network or keyboard inputs are marked as tainted, a status that is stored in a binary shadow variable associated with each register and every byte in physical memory. The virtualization system allows user-level applications within the domain to run at full speed on the unmodified hardware when they are operating on untainted data.

The hypervisor marks pages that contain shadowed data as unavailable, causing the system to take a page fault whenever instructions attempt to access them. The hypervisor's page fault handler then checks the page number against a list of pages that contain shadowed values. If the

process is attempting to access one of these pages, the entire state space of the domU domain is transferred into an instance of our modified version of the QEMU x86 emulator running within the dom0 administrative domain. The domU domain then begins executing within the emulator, which also performs taint analysis by propagating shadow values through copy and arithmetic operations. The emulator also checks the taint status of inputs to instructions such as dereferences and control flow operations.

When the emulator is no longer operating on tainted values, the domain's state is transferred back to the native hardware. Further information about this demand analysis system, including its method for detects tainted value and the overheads incurred by false sharing in the page table, can be found in the demand emulation paper by Ho *et al.* [14]. In this section, we describe the modifications needed to build a dataflow sampling system on top of this demand analysis framework.

We define performance overhead as the amount of time spent in analysis versus the amount of time running natively. Based on this definition, dynamic overhead values can be estimated as $(cycles\_analysis)/(cycles\_analysis + cycles\_native)$. Overhead can range from 0 (no analysis, full speed) to 1 (always in analysis), inclusive. Though this does not tell us the actual slowdown experienced by the system, we can pessimistically estimate the overhead experienced by the user by assuming that the analysis system makes no forward progress. In actuality, the performance can vary greatly, with an analysis system causing an average slowdown of $150\times$. Our no-forward-progress estimate, however, allows us to quickly calculate a relatively accurate estimate of dynamic slowdown. If the overhead is 0.95, the actual slowdown (assuming the analysis system slows performance by $150\times$) is $17.8\times$. Our estimate yields a pessimistic but close value of $20\times$, assuming that only 5 out of every 100 cycles make any forward progress.

This overhead calculation is done using a rolling window. The length of this window is user-defined, and each element in a window tracks the number of cycles spent within QEMU and in native execution during the last tick of the 100Hz system clock. The cycles stored in each element are updated both when a domain or QEMU is scheduled and on clock ticks. Finally, on every clock tick, the oldest unit is dropped from the window. This allows us to efficiently calculate the overhead for the last $window\_length * clock\_tick\_length$ period of time. The primary extensions to the baseline system to support this overhead tracking method include the addition of timer windows into the hypervisor and dedicated timekeeping code in the timer interrupt handler and the scheduler code of dom0 and the hypervisor.

We constructed a program that allows an administrator in the dom0 domain to set the overhead threshold for any domU. Similar to the overhead manager (OHM) described in QVM [2], our system then uses a daemon running in dom0 to check each domain's overhead against its threshold. If the overhead is higher than this limit, the OHM marks the domain as "above threshold" and sets a probability for the emulator resume native execution. This is the Bernoulli probability from Figure 4, and it can also be changed by the administrative program in dom0.

QEMU periodically checks the overhead-related variables stored by the OHM, and it probabilistically decides to stop the analysis based on the stored Bernoulli probability if the domain is "above threshold". If the emulator determines that it must stop, it clears all registers of their shadow values and returns control back to the hypervisor, where it resumes running normally.

While leaving the emulator clears the shadow values associated with the data in the registers, we also require the ability to clear meta-data associated with memory locations. Our page fault handler also checks the values set by the OHM before moving a domain into QEMU. If the overhead is beyond the desired threshold, the handler probabilistically decides whether to enter the emulation or skip the analysis. If the analysis is skipped, the shadow tags associated with all data on the page are cleared, the page table entry is marked as available, and execution continues natively at full speed.

We first implemented a taint analysis system that follows the same propagation and checking rules at Ho, *et al.* described [14]. Shadow values represent tainted variables, so removing a shadow variable implies that a memory location contains trusted data. We also built a dynamic heap bounds checker, similar to the system described by Nethercote and Fitzhardinge [23]. When a shadow variable is removed, its associated memory location is assumed to be of unknown type, which raises no errors when it is dereferenced.

5. Experimental Analyses

The virtual machines used in our experimental system are Linux-based systems with modified 2.6.12.6-xen0 and 2.6.12.6-xenU kernels. Tests were executed on a 1.8GHz AMD Operton 144 with 1GB of RAM and a Broadcom BCM5703 gigabit Ethernet controller. For all experiments, 512MB of RAM were allocated to both dom0 and domU.

5.1. Benchmarks and Real-World Vulnerabilities

Our simplest test, shown as pseudocode in Figure 6, is a synthetic benchmark that tests if our sampling framework controls overheads while performing intensive dataflow

Benchmark Name	CVE Number (Error)	Error Description	Detection at 1% overhead
Apache	CVE-2007-0774 [21]	A stack overflow in Apache Tomcat JK Connector v 1.2.20	Detected on every analysis
Eggdrop	CVE-2007-2807 [22]	A stack overflow in the Eggdrop IRC bot v 1.6.18	Detected on every analysis
Lynx	CVE-2005-3120 [19]	A stack overflow of the Lynx web browser v 2.8.6	Detected on every analysis
ProFTPD	CVE-2006-6170 [20]	A heap smashing attack of ProFTPD Server v 1.3.0a	Detected on every analysis
Squid	CVE-2002-0068 [18]	A heap smashing attack of the Squid proxy v 2.4.DEVEL4	Detected on every analysis

Table 1: Security Benchmarks. This lists a series of applications with known security exploits. The last column refers to the ability of the sampling taint analysis system to observe the exploits with an overhead threshold of 1%.

analyses. It takes in shadowed data and repeatedly performs computations on it, forming a single large dynamic dataflow. Without sampling, this program is constantly analyzed.

We also incorporated the *netcat* and *ssh* network throughput benchmarks from Ho, *et al.* to more directly compare dataflow sampling to a demand analysis system [14]. In their work, demand taint analysis experienced orders-of-magnitude slowdowns for applications that operated on large network transfers. In *netcat_receive* we move 1GB data from an external machine using a simple TCP connection. Both *ssh_receive* and *ssh_transmit* are similar, but instead move 781MB over an encrypted connection. We skip *netcat_transmit* because the only shadowed data it accesses is the IP address response from a DNS query. Our dataflow sampling system quickly clears this shadow value and the benchmark maintains high throughput until its completion.

We also analyze a collection of network-based benchmarks that suffer from security exploits that allow remote code execution. We use these programs to verify that we can observe a sufficient fraction of the dataflows in complex programs to find real-world errors even when sampling with low overhead thresholds. We ran these programs within our sampling system and transmitted exploits at random points in time in order to obtain high confidence in our error-finding capabilities. These programs are listed in Table 1. We focus on buffer overflows (that result in stack and heap smashing attacks) because the dataflow analysis we utilize in our prototype is designed to detect this class of errors. While different dataflow analysis systems may be better suited for other types of software errors, their specific design is orthogonal to this work on sampling mechanisms.

```
loop
  value ⇐ shadowed_data
  for i = 0 to N do
    value ⇐ value + 1
  end for
end loop
```

Figure 6: Worst-case Synthetic Benchmark. This program is designed to show the limits of our dataflow sampling system by continually operating on shadowed data. It will remain in emulation until the sampling system removes the shadow value.

5.2. Controlling Dataflow Analysis Overheads

Limiting Overheads in the Synthetic Benchmark. Our first set of experiments uses the synthetic benchmark detailed in Figure 6 to plot the instantaneous runtime overhead of our taint analysis tool. The program executes its conditional loop normally for 30 seconds leading to a computational throughput of about 1,800 million instructions per second (MIPS). Afterwards, shadowed data arrives from the network at a rate of one packet every five seconds. The content of the incoming packets is used for the computation within the loop, so our taint analysis system invokes the analysis mode. Our sampling system for this experiment uses a 30 second long overhead window, with a 100% probability of leaving analysis after crossing the overhead threshold.

As soon as packets begin to arrive, the performance of the non-sampling system plummets to an average of 21.3 MIPS. The system never leaves emulation when sampling is disabled because nearly every operation following the arrival of the first packet is shadowed.

Figure 7a shows the performance of our system when sampling is enabled with an overhead threshold of 10%. While instantaneous performance still drops when the first packet arrives, the overhead manager periodically forces the taint value to be cleared, returning the performance to its original value. This pattern repeats itself every time the rolling overhead window drops the short interval of low performance, resulting in an average performance loss of 11%.

Figure 7b plots both average slowdown and size of the analyzed dataflow over a range of threshold values. This shows that our sampling system allows users to effectively control the amount of time spent in dynamic analysis, although it may limit the number of dataflows that can be completely analyzed in one execution.

Performance Impacts for Real Benchmarks. We used network-intensive benchmarks to test the ability of our dataflow sampling technique to improve performance over the demand analysis system. Because network data is the source of shadow values in our system, demand analysis

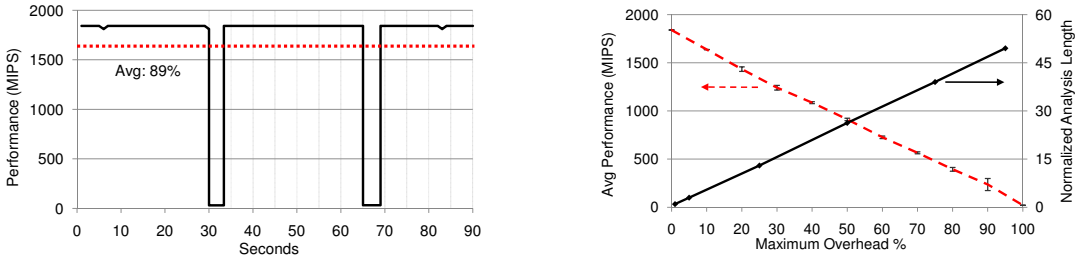


Figure 7: Synthetic Benchmarks: Performance and Analysis Accuracy. **(a)** Instantaneous (solid) and average (dashed) after the first untrusted packet arrives with a user-set overhead threshold of 10% **(b)** Average performance (dashed) and the maximum observable dataflow length normalized to a 1% overhead threshold (solid) for a range of overhead thresholds. Error bars show 95% confidence intervals.

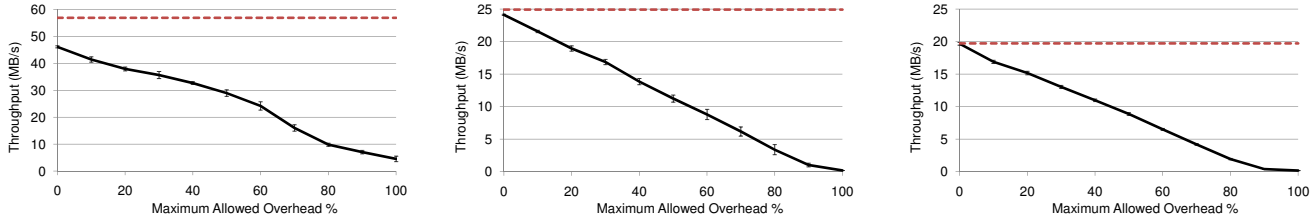


Figure 8: Dataflow Sampling Increases Network Throughput. The dashed red line represents performance with analysis disabled. Error bars represent 95% confidence intervals. **(a)** Netcat receive throughput increases nearly linearly as the maximum overhead threshold is lowered. The maximum performance is below that of a system with analysis disabled because of the page faults incurred to clear shadow data. Both **(b)** SSH transmit and **(c)** SSH receive allow nearly linear control over their overheads.

still causes the throughput (*i.e.* performance) of these programs to drop precipitously. For example, *ssh_transmit* suffers a 131× slowdown on a non-sampled system.

Figure 8 shows the throughput of these network benchmarks with dataflow sampling. The X-axis in each chart represents the maximum overhead threshold. The probability of leaving the analysis after crossing the threshold is set to 100%. The solid line plots average network throughput at that threshold, while the dashed red line represents the highest throughput each benchmark could reach with analysis disabled. Note that an overhead threshold of 100% is the same as a demand analysis system.

This data shows that dataflow sampling can greatly improve network throughput even for extremely slow analyses. Without sampling, for instance, transmitting data over *SSH* has an average throughput of only 171KB/s. This is 116× slower than the no-analysis case. However, as Figure 8b shows, we can directly control throughput by removing shadow values after the overhead threshold is reached.

5.3. Accuracy of Sampling Taint Analysis

Next, we test the dataflow sampling taint system on a number of programs to demonstrate that it can find exploits in real programs. We also verify that we can effectively analyze these programs, even when the maximum overhead is low. Finally, because programs that do not directly contribute to an error still require analysis and limit the time our erroneous process can be under analysis, we run a second set of tests with significant amounts of spurious system load.

Accuracy for a Lightly Loaded System. We ran the five programs shown in Table 1 in our sampling system with an overhead threshold of 1% in a 10 second window and a 100% probability of removing shadow values after reaching the threshold. For parameters as unforgiving as these, we were still able to find the security faults caused by the vulnerabilities in these benchmarks ***on every attempt***, as indicated in Table 1.

Though we cannot guarantee this is true for all security flaws, we have found that that vulnerabilities that appear early in the dataflow of a program are easier to exploit. Exploit writers must build their inputs such that every operation upon them modifies them to be in the correct form and place to attack systems. Intuitively, the fewer operations between the input and the final destination, the easier these inputs are to build. Errors that take place earlier in the dynamic dataflows are also easier to catch using sampling.

Accuracy for a Heavily Loaded System. The previous experiments focused on the analysis quality for benchmarks

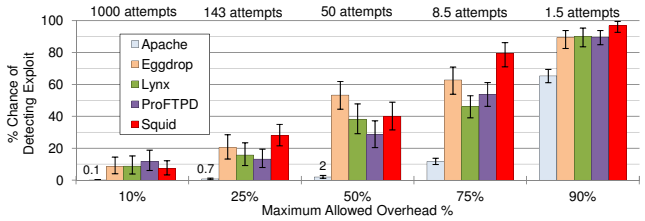


Figure 9: Exploit Observation with SSH Background Execution. Each bar represents the probability of detecting a particular program exploit. The number above each set is the expected number of attempts (or users) needed to observe the most difficult error. Error bars are 95% confidence intervals.

that were executed in isolation; the only dataflows were caused by exploits communicating with the vulnerable applications. Servers in real-world situations are typically heavily-utilized, with a resulting increase in the number of dataflows that the analysis system must check. Most dataflows in these high-utilization environments do not lead to errors, but still require time within the analysis system. Because our sampling system only allows a limited amount of time in analysis before attempting to remove shadow dataflows, these extra analyses raise the probability of ignoring dataflows in erroneous programs. This will lower the probability of finding an error. To quantify this difference, we studied the probability of observing an exploit while the rest of the system was heavily loaded with benign dataflows.

We combined two benchmarks to simulate such an environment. A network throughput benchmark (*ssh_receive*) runs in the background to simulate a large amount of harmless operations that still require analysis, while one of our vulnerable security benchmarks (from Table 1) is exploited at some random point in time. We start each benchmark by running the throughput program alone long enough to fill one overhead window. This allows us to avoid observing the exploit solely because the overhead window is not filled with background operations. The exploit for the second program is then sent after a random time interval; an exploit may arrive at any point within an overhead window.

All experiments were run with a 10 second overhead window and a 100% probability of leaving the analysis after crossing the overhead threshold. The 100% probability of stopping analysis means that the only dataflow randomization is caused by nondeterminism in the system and the random arrival time of the exploit. These are unforgiving settings that favor the user experience over analysis quality, as the short window and guaranteed halting of analysis yield short periods of time where any vulnerability can be observed and detected.

As Table 1 shows, our framework was able to consistently observe the exploits with an overhead threshold as low as 1% on the system with no benign dataflows. Figure 9 plots the probability of observing each benchmark's exploit on a heavily loaded system where *ssh_receive* runs in the background, causing a number of spurious dataflows that do not lead to any program flaws. The error bars represent the 95% confidence interval for finding the error, while the number of attempts needed to observe the most challenging exploit is listed above each overhead threshold. This can be thought of as the expected number of users required to run the analysis before detecting the error. Note that these attempts need not take place on the same computer.

The extraneous dataflows caused by *ssh_receive* make it more difficult to observe the exploit than in the case with no background execution. Rather than observing every exploit, a small number of exploits are missed at high overhead thresholds. At a 10% overhead threshold, we still observe most of the exploits nearly 10% of the time.

The Apache exploit is more difficult to observe, however. This benchmark moves a large amount of shadowed data numerous times before the exploit occurs, making the dataflow leading to the exploit relatively long and surrounded by other data that, while tainted, do not actually take part in the exploit. Our system still finds this flaw with a 0.1% probability at the lowest overhead threshold. This particular test shows that our system works well even with nearly every option set to particularly difficult choices. If we use *netcat_receive* instead of *ssh_receive*, for instance, the ability to observe the Apache exploit rises quite precipitously, as *ssh_receive* causes much more background analysis.

Changing the Probability of Removing Dataflows. We performed all our previous experiments with P_{st}, the probability of stopping dataflow analysis once we exceed the overhead threshold, set to 100%. This choice can obviously affect the performance of the system and its ability to fully observe large dataflows. A lower P_{st} permits some analyses over the overhead threshold to continue analysis before shadow values are cleared and analysis is forcibly stopped. While this gives us less control over the exact slowdowns a user perceives, it also allows some users in the population to observe more dataflows and analyze deeper into large dataflows. This boosts our chances of finding errors.

Figure 10 shows the performance of the system running our synthetic benchmark with a 25% overhead threshold and a variety of static P_{st} values. Setting P_{st} to 1% leads to a mean performance of 900 MIPS with a standard deviation of $\pm$170 MIPS. The mean is nearly the same as the performance attained with $P_{st} = 100\%$ at a 50% overhead threshold. However, because of the high standard deviation, more users will see a performance above 1070 MIPS, nearly the same as a user with $P_{st} = 100\%$ at a threshold of 40%. These particular users experience less overhead than they would with a threshold of 50%, while

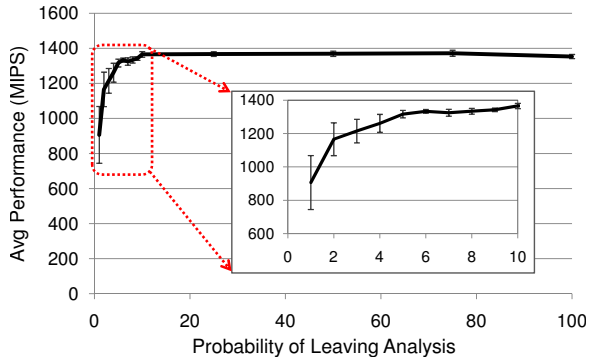


Figure 10: Changing the Probability of Removing Dataflows Affects Overhead. Low probabilities of removing meta-data once the OH threshold is crossed result in more performance variance; some users observe many more dataflows than others. Bars show standard deviation.

the analyses they cannot perform can be made up for by the users whose performance is more than one standard deviation lower.

This system may not be appealing to all users, but primarily serves as an example that modifying the sampling probability can significantly change both the overhead of a dataflow analysis and the probability of observing errors. We have analyzed a number of different mechanisms for setting this probability, from relating it to the amount of overhead beyond the threshold to connecting it to size of the dataflow currently under analysis. We have found that different mechanisms related to the size and number of dataflows work well for different benchmarks and analyses. Due to space limitations, we will detail this in-depth analysis in a future report.

6. Related Work

6.1. Dynamic Dataflow Analysis Systems

Nethercote and Seward built Valgrind, a popular dynamic binary instrumentation tool that is the basis for a number of dynamic dataflow tools because of its strong support for shadow memory [25], [24]. Memcheck is used to find runtime memory errors such as uses of undefined variables and memory leaks [30], while Annelid is the preliminary design of a dynamic bounds checker [23]. TaintCheck, one of the first software-only taint analysis systems, is implemented within Valgrind [26], as are both Helgrind and ThreadSanitizer, two dynamic data race detectors [29].

Many researchers focus on reducing the extremely high overheads associated with these dynamic dataflow analyses. Minos [9] and Raksha [10] are examples of taint analysis systems implemented in hardware, while HARD is an example of a hardware race detector [33]. More generally, MemTracker offers a programmable hardware state machine that allows simple dataflow analyses to be run at hardware speeds [31]. While these systems allow dataflow analyses with little runtime overhead, they require costly hardware changes and are not currently available on modern processors.

Some researchers have focused on changing the analysis algorithms to improve performance. Umbra attempts to reduce the overhead of accessing shadow variables, which should accelerate any dataflow analysis system [32]. Others have looked at ways to filter analyses that cannot possibly cause errors [28]. While these works yield better performance, they do not completely solve the overhead problem because many dataflow analyses must still perform many calculations alongside the original program.

Other researchers have therefore suggested decoupling the analysis from the original program. This allows them to, for instance, parallelize the analyses using multiple cores [27], [7], [8]. This method still causes a reduction in total throughput of the system because multiple cores are used to perform the analysis and are thus not available for other uses. Chow, *et al.* also describe a mode where they log specific dataflow information for later offline analysis. This mechanism allows for low-overhead execution on users' systems, but would be prohibitively expensive for a developer if every user were to send back numerous logs. If users only sent back logs for analysis when a crash occurred, some bugs may not be caught until after they caused crashes or were exploited. This defeats one of the major benefits of dataflow tests such as taint analysis or race detection: they can catch errors that have not yet caused a crash.

6.2. Dynamic Sampling Systems

One way to accelerate dynamic analyses is by using sampling. To the best of our knowledge, ours is the first system to enable software-based sampling of dynamic dataflow analyses on unmodified binaries. This section describes related sampling works and discusses how they fall short for dataflow checking.

Liblit, *et al.* showed that Arnold and Ryder's method, described in Section 2.3, is inadequate for statistically weighting bugs reported by users because of its deterministic nature. The system designed by Liblit, *et al.* instead has users perform checks with some stochastic probability and report any failed checks when a bug manifests [15]. Bursty Tracing quickly enables dynamic program tracing for profiling purposes, then disables it after a deterministic number of instructions [13]. Hauswirth and Chilimbi later looked at ways to use profiling feedback to focus their dynamic analyses on rarely-executed portions of the program [12]. Much like the works by Arnold and Ryder, these

code-based sampling methods are inadequate for dataflow analysis system. They can cause both false positives and extra false negatives because they do not appropriately handle shadow dataflows.

Arnold, *et al.* developed QVM, a Java virtual machine that uses sampling to reduce the slowdown of multiple introspective analyses. [2]. The strength of their sampling system relies on their ability to follow random objects from invocation until destruction. This would be difficult to do in a system with less access to language-level constructs than a Java virtual machine. Our dataflow sampling system works on unmodified binaries compiled from any language.

LiteRace describes a technique for performing sampling in a dynamic data race detector [17]. Though this is a form of dataflow analysis, their code-based sampling technique utilizes an idiosyncrasy of data race detection to avoid any false positives. Because the meta-data (vector clocks) only flow from synchronization points to individual variables where they are checked, LiteRace simply chooses not to skip any synchronization points in their sampling system. This means that their system only skips tests and thus will thus only incur false negatives. PACER utilizes a similar insight to decrease the rate of false negatives in a sampling race detector [6]. These methods will not work for other, more generalized dataflows analyses.

In contrast to the software-based works previously mentioned, our recent work, Testudo, is a hardware-based dataflow sampling system [11]. Its primary goal is to reduce the hardware storage required for shadow data, but it can also be used to sample other analyses at reduced overheads. Besides requiring non-trivial hardware additions, it is not designed to contain runtime overheads; this is a side-effect of its architecture, and it is not directly controllable at runtime.

7. Conclusion and Future Work

In this paper we presented the first software-based scalable distributed dataflow analysis system. We are also the first to use dataflow sampling to allow users to control the overheads they observe when running heavyweight dataflow analyses. This allows developers to distribute their software with dataflow analyses to large populations, allowing numerous users to test software for security, correctness and performance bugs. We also showed that it is possible to observe many real-world errors, even when users set their maximum overhead very low and strictly stop the analysis when they cross this threshold.

We are currently exploring a number of future research directions to this work. We believe that finding the best mechanisms for setting the Bernoulli probability of removing shadow variables could be a very fruitful research direction. Our preliminary results imply that this may be analysis-specific, due to the types of errors and possible dataflows shapes that must be observed to catch them.

Our current system requires that shadowed data be marked in the page table to indicate when it is accessed. There may be simple hardware modifications that allow our system to find this data at finer granularities.

Finally, we would like to extend our work by sampling in different ways. For example, instead of constantly performing analysis until the threshold is reached, which may lead the sampling system to throw away all analysis up to that point, it may be possible to observe ahead of time that a particular part of the program requires an unaffordable amount of analysis and report this fact to the developers for further investigation or offline analysis.

Acknowledgments

The authors wish to thank Andrea Pellegrini, Debapriya Chatterjee, and Mona Attariyan for their help with this work. We also wish to thank Steven Hand and Andrew Warfield for access to their demand taint analysis system and Robert Perricone for his time with the benchmarks. The authors acknowledge the support of the National Science Foundation and the Gigascale Systems Research Center.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2001.

[2] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *Proc. of the 23rd Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

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

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th Symp. on Operating Systems Principles*, 2003.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conf.*, 2005.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2010.

[7] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the 35th Int'l Symp. on Computer Architecture*, 2008.

[8] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of the USENIX Annual Technical Conf.*, 2008.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th Int'l Symp. on Microarchitecture*, 2004.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, 2007.

[11] 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 *Proc. of the 41st Int'l Symp. on Microarchitecture*, 2008.

[12] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2005.

[13] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *the 4th Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[14] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of the European Conf. on Computer Systems*, 2006.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2003.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[17] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2009.

[18] National Vulnerability Database. Vulnerability Summary for CVE-2002-0068: Squid 2.4 STABLE3 and earlier, 2002.

[19] National Vulnerability Database. Vulnerability Summary for CVE-2005-3120: Lynx 2.8.6, 2005.

[20] National Vulnerability Database. Vulnerability Summary for CVE-2006-6170: ProFTPD 1.3.0a and earlier, 2006.

[21] National Vulnerability Database. Vulnerability Summary for CVE-2007-0774: Apache Tomcat JK Web Server Connector 1.2.19/1.2.20, 2007.

[22] National Vulnerability Database. Vulnerability Summary for CVE-2007-2807: Eggdrop 1.6.18, 2007.

[23] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

[24] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proc. of the Int'l Conf. on Virtual Execution Environments*, 2007.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2007.

[26] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symp.*, 2005.

[27] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[28] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *the Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

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

[30] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conf.*, 2005.

[31] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *Proc. of the 13th Int'l Symp. on High-Performance Computer Architecture*, 2007.

[32] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2010.

[33] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware–assisted lockset-based race detection. In *Proc. of the 13th Int'l Symp. on High-Performance Computer Architecture*, 2007.