Testudo: Heavyweight Security Analysis via Statistical Sampling

Joseph L. Greathouse   Ilya Wagner   David A. Ramos   Gautam Bhatnagar
Todd Austin   Valeria Bertacco   Seth Pettie

Advanced Computer Architecture Lab
University of Michigan
Ann Arbor, MI
{jlgreath, iwagner, david.ramos, gautamb, austin, valeria, pettie}@umich.edu

Abstract

Heavyweight security analysis systems, such as taint analysis and dynamic type checking, are powerful technologies used to detect security vulnerabilities and software bugs. Traditional software implementations of these systems have high instrumentation overhead and suffer from significant performance impacts. To mitigate these slowdowns, a few hardware-assisted techniques have been recently proposed. However, these solutions incur a large memory overhead and require hardware platform support in the form of tagged memory systems and extended bus designs. Due to these costs and limitations, the deployment of heavyweight security analysis solutions is, as of today, limited to the research lab.

In this paper, we describe Testudo, a novel hardware approach to heavyweight security analysis that is based on statistical sampling of a program's dataflow. Our dynamic distributed debugging reduces the memory overhead to a small storage space by selectively sampling only a few tagged variables to analyze during any particular execution of the program. Our system requires only small hardware modifications: it adds a small sample cache to the main processor and extends the pipeline registers to propagate analysis tags. To gain high analysis coverage, we rely on a population of users to run the program, sampling a different random set of variables during each new run. We show that we can achieve high coverage analysis at virtually no performance impact, even with a reasonably-sized population of users. In addition, our approach even scales to heavyweight debugging techniques by keeping per-user runtime overheads low despite performing traditionally costly analyses. Moreover, the low hardware cost of our implementation allows it to be easily distributed across large user populations, leading to a higher level of security analysis coverage than previously.

1. Introduction

The computer industry bears a significant cost from software bugs, primarily because they are the basis for most security vulnerabilities. A recent survey of CERT advisories found that 67% of these vulnerabilities were caused by software bugs, which attackers could utilize to remotely gain control of a program [2]. Many techniques that enable an external attacker to take advantage of a software bug require the manipulation of external inputs (*e.g.*, network, file I/O) to force the program into performing an unintended operation. A classic example of this type of security vulnerability is the buffer overflow attack, where an array access bug is used to overwrite a function's return address on the stack, causing a jump to malicious code. An SQL injection attack is another example of a security vulnerability whereby an attacker injects SQL code into a server-generated query, thereby gaining unauthorized access to a back-end database.

A variety of technologies have been developed to detect and prevent security vulnerabilities. *Array bounds checking* is a popular technique that prevents the often-exploited buffer overflow attack [29]. Researchers have recently proposed *dynamic information flow tracking (DIFT)* techniques in an effort to preemptively curtail vulnerabilities [28]. The goal of DIFT techniques is to detect exploitable bugs by adding instrumentation to a software application to identify dataflows resulting from insufficiently validated external inputs. Examples of DIFT techniques include taint checking [23] and input bounds checking [11]. DIFT techniques are more powerful than prevention techniques, such as array bounds checking, because they can detect security vulnerabilities without needing an active attack on the program, allowing developers to protect their programs before any attack is attempted.

DIFT works by tagging external data values entering the system at the application's I/O interface (*e.g.*, user inputs, disk accesses, *etc.*) with tag values specific to the analysis under study. Tags are then propagated during program execution to other variables derived from the external data. Finally, tag values are used to determine the presence of bugs that could lead to security vulnerabilities when a program's variables are used in potentially dangerous operations. For example in *taint checking*, a taint bit (tag) is attached to all external input values. As the program runs, all computations check whether any of their inputs are tainted, and if so, the result is also marked tainted. A datum remains tainted until it is an operand in a relational operation, after which it is considered "checked" and safe to use. A security vulnerability is exposed when a tainted data value is used to access memory (*e.g.*, as a base register in a memory access) or redirect program control (*e.g.*, as an input to an indirect jump).

Taint analysis techniques assume that checks in the program (implemented as relational tests on data values) are always sufficient to ensure that values are correct; however, escaped-bug analyses have not always shown this to be the case [4, 11]. Many security vulnerabilities are the result of input checks that are insufficiently constrained or outright incorrect. Consequently, more powerful DIFT techniques, such as *input bounds checking*, can be used to validate the bounds checks applied to external inputs [11]. This approach works by attaching symbolic expressions to all externally derived data. These expressions specify the possible range of input data, which is inferred from predicates applied to each externally derived input during program execution. For example, if the program flow passes a branch instruction in which an external value x was tested to be less than 5, then it is reasonable to infer that, for all program executions, the external datum x satisfies the constraint $x < 5$ at that point in the program. When potentially dangerous operations are executed, the symbolic expression is checked to verify that derived constraints are sufficiently tight to prevent buffer overruns and incorrect control transfers. The approach is very powerful in ensuring that externally derived inputs are correctly bounded and validated by the program.

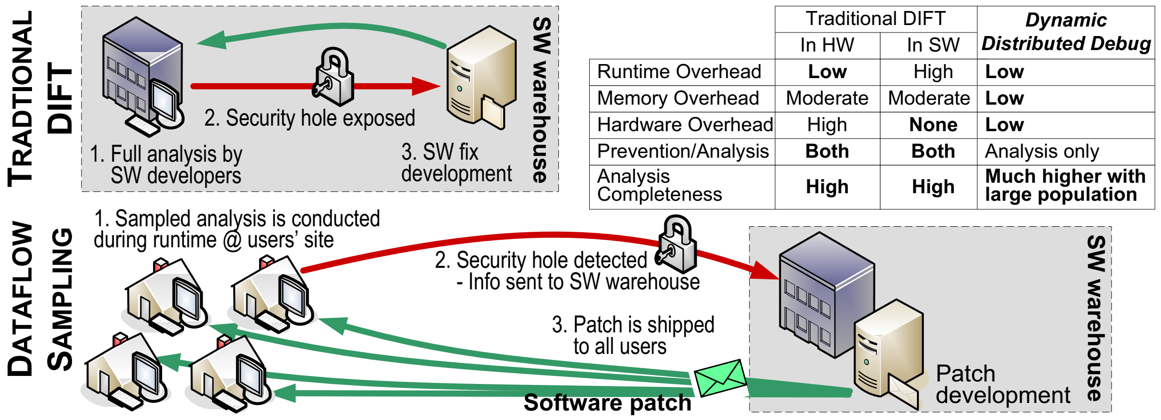
DIFT techniques in general are extremely powerful and can expose many security vulnerabilities, but their use incurs a high cost. Software implementations suffer performance slowdowns of 2-3x in the best case for taint checking [28] and 13-220x slowdown for very heavyweight analysis. These slowdowns are the result of the significant software instrumentation needed to record, propagate and check value properties. To mitigate this overhead, researchers have proposed hardware techniques that improve the performance of security analysis [5, 6, 32, 33]. In general, these techniques add storage space in the memory system for the tag bits and extend the processor pipeline to compute, propagate and check tagged values [28]. Unfortunately, these extensions lead to notable memory overheads, in many cases approximately one bit per byte [6]. Even more dramatic are their design costs: a large number of components must be modified and enhanced to support this technology, including memory systems, buses, pipelines, and I/O devices. Finally, hardware support to date has only been proposed for taint checking, and it is unlikely that it could be extended to more heavyweight analyses, such as input bounds checking, without incurring significant cost burdens.

1.1. Contributions of This Paper

To date, the high overhead of DIFT techniques have limited their use to, at best, debugging in the software development lab, where developers and test engineers exercise a software application under development to expose security vulnerabilities. Unfortunately, the low performance of DIFT analysis and the testers' inability to accurately predict how users will exercise programs has led to a myopic search for software vulnerabilities, as evident by the high number of bugs that escape the testing process. To overcome this critical limitation, the industry must adopt a much more scalable approach to security vulnerability analysis, ideally one in which actual users perform analyses in their day-to-day activities. Users constitute a large population of potential testers whose activities and test environments best represent how a program is used, and thus how it could be exploited. *The largest challenge, addressed with this work, is to deliver high-coverage heavyweight security vulnerability analysis solutions to a user base that cannot tolerate slowdowns of more than a few percent.*

In this paper, we propose a *dynamic distributed debugging* technique, capable of continuous in-field security vulnerability analysis of programs. Our goal is to leverage a large user population to find those vulnerabilities which have slipped through in-house testing. Our solution, called Testudo, employs a "sample and conquer" approach to provide high-coverage analysis with very low performance overhead for individual users. To limit overheads, the approach periodically samples potentially vulnerable program dataflows and analyzes only a small, randomly-selected subset of the externally derived variables. Our solution incorporates into the hardware architecture a small *sample cache* (whose size can be designer selected) that tracks the dataflow of a single external variable from its inception and through its descendants. Externally derived values enter the sample cache using a feedback-controlled lottery system, which limits both the performance and memory overheads associated with the specific DIFT analysis. To gain the high coverage necessary to eliminate security flaws, the statistical selection process is designed to track (with high probability) different variables for each program execution. Consequently, multiple runs of the program (or many single runs on multiple machines) will eventually combine to provide a complete program analysis. By tuning the number of external values tracked during each execution (possibly down to a small fraction of a single dataflow), we can arbitrarily limit memory and performance overhead, at the expense of needing more cumulative runs to achieve full-coverage analysis. Fortunately, programs that are common targets of security attacks (*e.g.*, Apache, SSH, Windows) typically have a large user base. Thus, we can incur minimal overhead and still achieve very high analysis coverage.

Figure 1 compares Testudo to traditional DIFT techniques. Testudo has the advantage of having very low hardware cost and run-time overhead compared to traditional software- or hardware-based checking technologies. Additionally, the design impact of our approach is limited to the inclusion of a sample cache and DIFT analysis logic into the processor pipeline. No other modifications to the hardware are required. Unlike traditional DIFT analysis, which could only be run on a few machines in the test lab, our dataflow sampling approach scales the analysis to many users' machines. Testudo utilizes a statistical sampling solution when tracking


TRADTIONAL DIFT
2. Security hole exposed
1. Full analysis by SW developers
3. SW fix development
SW warehouse
DATAFLOW SAMPLING
1. Sampled analysis is conducted during runtime @ users' site
2. Security hole detected - Info sent to SW warehouse
3. Patch is shipped to all users
Software patch
Patch development
SW warehouse

	Traditional DIFT		*Dynamic Distributed Debug*
	In HW	In SW	
Runtime Overhead	**Low**	High	**Low**
Memory Overhead	Moderate	Moderate	**Low**
Hardware Overhead	High	**None**	**Low**
Prevention/Analysis	**Both**	**Both**	Analysis only
Analysis Completeness	**High**	**High**	**Much higher with large population**

Figure 1: Traditional DIFT analysis vs. Testudo. Traditional DIFT analysis is carried out at the software development site and incurs high performance / memory overhead and hardware costs. Developers rely on it to identify and resolve security bugs before an application is released. In contrast, Testudo is based on on a statistical dataflow sampling approach; it combines the results of many individual-user partial analyses and achieve high coverage. Hardware and performance overheads are minimal, allowing Testudo to be deployed at the user site. Each user collects partial security vulnerability analysis results during each use of a program; when and if a potential exploit is found, the relevant information is sent back to the software developer for patch development.

externally derived dataflows. Doing so ensures that memory and performance overhead remains much lower than that of traditional DIFT techniques, and enables a large user base to provide unprecedented levels of security vulnerability coverage. With Testudo, any vulnerability detected in the field is transmitted back to the software warehouse (using system-software channels), where a patch is developed and distributed back to customers. Over time, our high-quality security vulnerability analysis will excise all security vulnerabilities with virtually no performance impact perceived by the user.

The remainder of this paper introduces our dataflow sampling technology and analyzes its cost, performance and analysis coverage. Section 2 introduces the technology and shows how it can be applied to security analysis. Section 3 details our experimental framework and provides analytical and empirical coverage results for various sample cache sizes, applications and user populations. Section 4 covers related work in the area, and Section 5 concludes and outlines future research directions. Finally, an appendix at the end of the paper presents an analytical model that relates population size with analysis coverage.

2. Hardware for Dynamic Distributed Debug

The overall goal of DIFT analysis is to tag externally derived inputs, track their progression through the system, and finally, validate their use in potentially dangerous operations. In this section, we detail our hardware support for dynamic distributed debugging.

2.1. Baseline Support for DIFT Analysis

To implement DIFT analysis, program execution semantics must be extended to support the capability to i) tag program values that originate from external inputs, ii) monitor the dataflows of externally derived values in order to propagate tags, iii) detect checks on externally derived values for the purpose of validating external inputs, and iv) detect when a tagged variable is used to perform a potentially dangerous operation such as a memory access or control transfer. An example of our DIFT approach is shown in Figure 2.

Variables are *tagged* by the DIFT analysis system when an external input is loaded into a memory location or register. We utilize a special "tagged load" instruction in our design, which device drivers can employ to mark incoming data from an I/O device or DMA buffer. Use of the tagged load indicates to the system that the input value is from an external source and that it should undergo DIFT analysis. All storage locations conceptually support a tag bit, indicating that the value is being tracked for DIFT analysis. As shown in Figure 3, Testudo enhances each entry of the register file with an extra tag bit; tagged load instructions and memory loads of tagged data set this bit in the corresponding register file entry. Tag bits in the register file are propagated down the pipeline so that instruction results can also be tagged. In previously proposed hardware tagging schemes, cache lines and memory locations were also extended to accommodate the extra tag information. However, as we detail in Section 2.2, Testudo consolidates all tag information into a single *sample cache* located within the processor pipeline.

If a tagged value is used in a potentially dangerous operation (*e.g.*, as the base register of a load or store) or as the destination address for an indirect jump, Testudo will by default declare a security violation trap. Once the security trap handler is invoked, it has great flexibility: it can validate the action based on its input, send error-report information to the software development warehouse, or even terminate the program to prevent malicious operations.

The baseline DIFT analysis semantics simply propagate value tags to the results of instructions. To adapt to a broad range of security analyses, Testudo offers a policy enhancement mechanism that can be tailored to many DIFT analysis semantics. Different DIFT analyses can be implemented through the use of the *policy map*, shown in Figure 3.

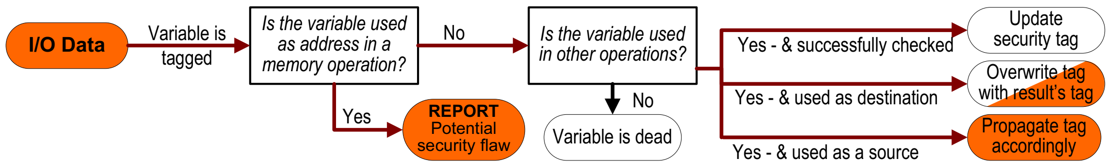

I/O Data
Variable is tagged
Is the variable used as address in a memory operation?
Yes
REPORT Potential security flaw
No
Is the variable used in other operations?
No
Variable is dead
Yes - & successfully checked
Update security tag
Yes - & used as destination
Overwrite tag with result's tag
Yes - & used as a source
Propagate tag accordingly

Figure 2: The life of a tracked variable. Data brought in from an external source is tagged as potentially unsafe. Subsequent operations on that data remove the tag or propagate it to other variables based on security analysis-specific rules. When a variable is overwritten by an operation's result, it inherits the tag of the operation's result. If potentially dangerous operations are performed on tagged data, a warning is flagged.

Using the policy map, system software can install pointers to handler routines that are invoked upon relevant events such as loading or storing tagged data, overwriting tagged data, executing specific operations on tagged data, *etc.* Within a security handler, it is possible to set or clear tags, inspect arbitrary program state, and access private bookkeeping data in a non-invasive manner. Through these policy handlers, it becomes possible to implement a variety of DIFT policies.

For example, to implement a taint analysis system [6], one only needs to install handlers to reset tag (taint) bits when a tagged value is an input to a relational operator, since after this operation the value is considered untainted. In addition, a security trap handler must be called when values are loaded with a tagged base register or when an indirect jump has a tagged operand. The security trap records the occurrence of the bug, and it could be used to transmit debugging information back to the developers.

Another example of a DIFT analysis that can be implemented using Testudo is input bounds checking [11]. This heavyweight analysis requires the creation of a symbolic interval constraint expression for each externally derived variable, to indicate its valid value bounds. By installing service handlers when tagged values are loaded, stored, operated upon and overwritten, it is possible to accurately compute the constraints as detailed in the original software-based solution. An additional handler calls a constraint solver on potentially dangerous operations to determine if the tagged value belongs to an interval that could allow an illegal operation (even if the actual tagged value is legal). Using this approach, it becomes possible to fully validate bound checks applied to the externally derived values. However, with this additional power, input bound checking analysis becomes expensive, approximately 50 instructions of service handler code for each instruction accessing a tagged value. The original software-based implementation of this approach resulted in a 200x slowdown for programs with frequent externally derived dataflows. To tackle these situations, Testudo is equipped to carefully control the slowdown experienced by a user through the use of a dataflow sampling technique that reduces performance and memory overheads.

2.2. Limiting Performance and Memory Overheads

Our overarching goal is to build a system that can achieve high coverage DIFT analysis at extremely low hardware and performance costs by distributing the analysis workload over a large population of machines. To minimize the impact to users and thus encourage widespread use of the analysis mechanism, we must devise techniques to limit both memory and performance overheads of the analysis. It is important to note that the hardware modifications that Testudo requires are limited to the processor pipeline. Thus, no changes or costs are incurred in caches, the memory system, or buses.

Sample cache design. Traditional hardware-based DIFT techniques (*e.g.*, taint checking) add one tag bit per memory byte to track externally derived variables through memory. This memory overhead also results in considerable design cost as caches, memory interfaces and interconnect buses must be extended to support tag bits. However, *we observe that DIFT analysis for a single path in a dataflow can be implemented with a single entry of address-configurable tag storage*. It follows from this observation that, if we execute a program many times, each time selecting a different path in a dataflow and/or selecting different dataflows in the program, we eventually analyze the full program while strictly limiting memory and design costs. If additional tag storage entries are available, we can examine multiple paths and dataflows in a single run of the program. Thus, fewer executions will be required to explore all potential security vulnerabilities. Similarly, we can limit the performance impact of DIFT analysis code by limiting the rate at which we analyze new dataflows and paths, again at the cost of more executions necessary to analyze the program. *Thus, there exists a convenient trade-off between DIFT analysis cost (performance and memory overheads) and the number of executions required to achieve full-coverage DIFT analysis.*

As shown in Figure 3, we limit the cost of tag storage using a small physically tagged cache, which we call the *sample cache*. The sample cache holds the physical addresses of currently tracked memory variables. For the most part, the sample cache behaves as any other cache, except that i) it does not contain data storage, since valid bits and physical tags are sufficient to denote that a memory variable is being tracked, and ii) it uses randomized replacement and insertion policies to ensure good coverage of dataflows and to limit the DIFT analysis's performance impact. To ensure that we visit as much of each tracked dataflow as possible, we only insert *one* dataflow into the sample cache at a time. By employing this policy, it becomes possible to see all of a large, long-lived dataflow, which would otherwise be quickly displaced by newly tagged nodes from other dataflows. It is trivial to implement this single dataflow management policy by i) only inserting into the sample cache tagged memory addresses

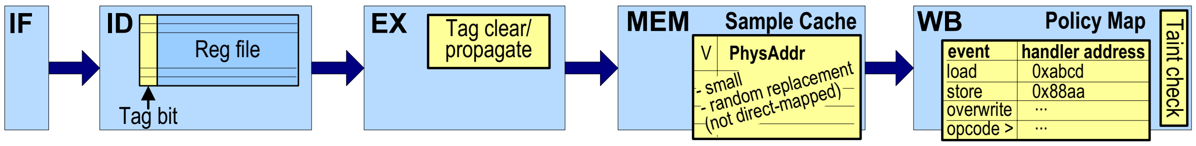


Figure 3: Testudo hardware implementation. To support sampled DIFT analysis, we add a small amount of hardware to the existing processor, as highlighted in the schematic. A tag bit is added to each register in the register file to track externally-derived values. Simple tag propagation logic is placed along the pipeline to handle setting and clearing of tags. The sample cache is a small cache used to monitor a handful of physical memory addresses that are currently tagged. The policy map enables a broad range of user-specified DIFT analysis semantics through the use of service handlers, invoked at specific analysis events.

that are derived from tagged values currently in the sample cache, and ii) flushing the entire contents of the sample cache when a tagged memory address from another tainted dataflow is introduced into the sample cache. In addition, Testudo implements sample cache replacement policies within a single dataflow (intra-flow policy) and between different dataflows occurring within a program execution (inter-flow policy).

Intra-flow selection policy. While only one tagged dataflow resides in the sample cache at a time, it still may be the case that the currently tracked dataflow is larger than what can be stored in the available sample cache resources. Consequently, the intra-flow policy must implement a replacement scheme that ensures high coverage analysis of large dataflows across multiple executions. Note that any traditional deterministic cache replacement mechanism, such as least-recently-used (LRU), would not work for the sample cache: in fact, any deterministic cache replacement policy would result in the same tagged dataflow values being examined in subsequent runs of the program (given the same program inputs). Consequently, we have adopted a random replacement policy, whereby for each candidate replacement, all the resident cache entries as well as the newly arrived tagged address have equal probability of being evicted from the cache.

Moreover, for DIFT analyses with high service handler costs, the chance of entering the sample cache can be reduced much more sharply. In these situations, instead of uniformly random eviction, Testudo assigns higher probability of exclusion from the cache to the incoming variable, since there is a higher chance that the new variable has not been visited before, thus triggering even more handler-serviced events. This biasing is adjusted dynamically throughout a program's execution: Testudo monitors the frequency of analysis invocations as well as the average analysis cost per invocation, altering the eviction probabilities to keep the performance penalty at a constant, user-acceptable level. Thus, if service handlers begin to be triggered more often or begin to incur higher penalty, newer variables are disregarded with higher probability. In hardware, this dynamically adjusted random selection is implemented with a weighted random number generator [15], in as little as $2 * \log(cache\_size)$ LFSR bits and $\log(cache\_size)+1$ weight bits. In Section 3, we start to explore the impact of specialized fine-tuned biasing of intra-flow random replacement policies as a means to achieve high coverage with even fewer executions.

Inter-flow selection policy. The last component of Testudo's cache replacement policy is how to select which newly created external input variables (dataflow roots) to track in the sample cache. This inter-flow policy is dynamically adjusted based on the rate of arrival of new dataflows and the constraint on perceived performance impact. Each new dataflow, created when an input arrives or a load from I/O buffer occurs, polls the interflow selection mechanism to determine whether it can overwrite the dataflow currently being analyzed by the sample cache. To ensure high coverage of the dataflow currently in the sample cache, the probability of a flow replacement is set quite low. We conceptually partition the execution into epochs: for each epoch the number of challenges to replace the dataflow under analysis is fixed (say, n). We then enforce two rules: i) each challenger has a small probability to overwrite the current flow ($0.1 \times 1/n$ in our experiments), and ii) at most one challenge can be won during an epoch. The chance of entering the sample cache can further decrease if the performance impact of the currently tracked dataflow rises above the performance impact allowed by the application developers (typically 5% or less). For very heavyweight analyses, the tracked values may degrade to a single path of a single dataflow, thus ensuring forward progress toward the complete analysis of the program.

Example. Figure 4 shows the dataflow coverage achieved for one of our experimental benchmarks, *tiff.bad*, by using the replacement policies discussed above over multiple executions. In this example, we show Testudo performing simple taint analysis while tracking only three of the dataflows of this benchmark. Hence, the service handlers have virtually no performance impact, and thus no back-off adjustment is required in the replacement probabilities. The figure shows the coverage attained after 10, 25, 100 and 250 executions of the program. Each node in the dataflow is colored with a different shade of gray, based on how many times it has been covered cumulatively (0=white, 10+=black). In this example the sample cache is a 4-entry fully associative cache and Testudo could achieve full coverage after 231 executions.

3. Experimental Evaluation

In this section, we detail the framework and benchmarks that we used for the experimental evaluation of Testudo. We then detail our findings in using Testudo as a taint analysis framework by reporting the performance of the solution and user population required to achieve 99% program

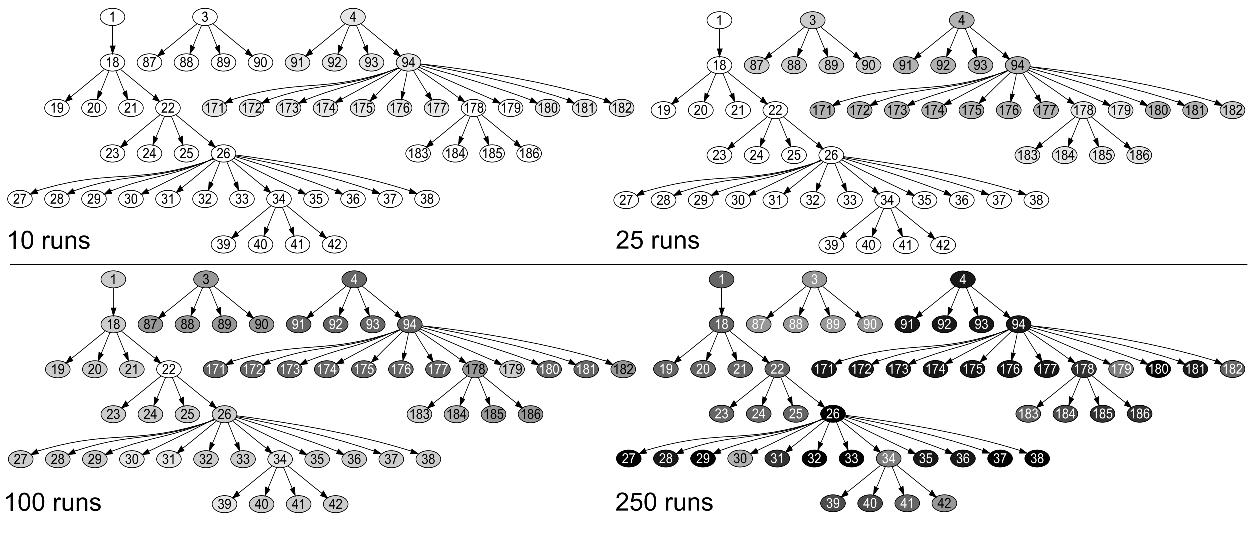


Figure 4: Coverage of 3 dataflows from the tiff.bad benchmark. Node labels indicate the chronological order in which each tainted variable arrives from the pipeline to the sample cache. Coverage results are shown after 10, 25, 100 and 250 runs on a system with a 4-entry fully-associative sample cache, where different shades of gray indicate how many times each node has been covered cumulatively (0=white, 10+=black).

dataflow coverage. We compare these coverage results with the analytical upper bound computed using the worst case analysis technique described in the appendix. Finally, we study applications beyond taint analysis and evaluate the performance overhead of Testudo for a range of execution overheads in the service handler routines. We conclude this section by reporting the area, power and access time of the sample cache sizes we used.

3.1. System Simulation Framework

To evaluate the performance of a taint analysis system based on Testudo, we implemented a *data tracking simulator*, which generates dataflows for each of the experimental benchmarks, and a *sample cache simulator*, which analyzes these dataflows for coverage. The data tracking simulator is implemented on Virtutech Simics [16], a full-system functional simulator able to run unmodified operating systems and binaries for a number of target architectures. We targeted an Intel Pentium 4 system running Fedora 5, and augmented Simics with a module to track tagged memory addresses, general purpose registers, status/control flags, and segment registers. This module is implemented using code from the Bochs IA-32 emulator [1], and uses tag propagation and clearing techniques similar to [2, 28]. Tagged variables are tracked with 32-bit word granularity, as in [6]. All external data coming from disk, network and keyboard is tagged on arrival. The simulator only tracks dataflows relevant to the benchmark under study, by monitoring the Linux kernel and the relevant Linux process IDs. The traces produced by the tag tracker are then analyzed by the sample cache simulator.

3.2. Benchmarks

We chose a set of ten benchmarks consisting of popular Linux applications with known exploits. Table 1 summarizes the relevant statistics of these benchmarks, including program execution length (in cycles), number of tainted dataflows (*i.e.*, external variables from untrusted sources) and total number of tagged words to be tracked. We also list the total number of unique addresses spanned by the tracked variables (two variables may share an address if they have non-intersecting life spans, a common situation for stack variables). Finally, the table reports the size of the largest dataflow. Note that single node dataflows cannot occur, since a dataflow's root comes from I/O space. This value will be explicitly tainted by the I/O driver.

- **Libtiff** - (*tiff.good, tiff.bad*). Library providing TIFF image manipulation services. Versions 3.6.1 and earlier fail to check bounds on RLE decoding [8]. The benchmarks parse a valid and an invalid image.
- **Xpdf** - (*xpdf.good, xpdf.bad*). PDF document handling library. Versions 3.0.0 and earlier are vulnerable to a PDF format issue [30]: a malicious PDF file could contain invalid tree node references that would never be checked. The two variants parse a valid and an invalid PDF.
- **Eggdrop IRC bot** - (*eggdrop.good, eggdrop.bad*). Open source IRC bot. Version 1.6.18 fails to check bounds on data supplied by an IRC daemon, leaving it vulnerable to buffer overflow attacks [27]. This benchmark receives both benign and malicious data from an IRC daemon.
- **Telnet** - (*telnet.server, telnet.client*). A popular but insecure remote connection server. The benchmark client connects to a benchmark server and logs in to a user account.
- **Lynx web browser** - (*lynx*). Text-based web browser for command line environments. Versions 2.8.6 and earlier fail to properly check bounds on NNTP article headers [26]. The benchmark sends a malicious article header from a simulated NNTP server to the Lynx browser.
- **Apache server with PHP** - (*httpd*): The Apache web server and PHP have experienced numerous public ex-

	tiff.good	tiff.bad	xpdf.good	xpdf.bad	eggdrop.good
Total Cycles	2,600,000	3,500,000	15,000,000	15,000,000	114,000,000
Tracked Words / Unique Addresses	987 / 299	214 / 68	195,853 / 5,729	73,613 / 3,771	3,854 / 1,232
Largest Dataflow Words / Addresses	116 / 29	59 / 15	185,465 / 2,817	64,015 / 316	678 / 25
Number of Dataflows	71	17	540	960	150

	eggdrop.bad	telnet.server	telnet.client	lynx	httpd
Total Cycles	245,000,000	90,000,000	50,000,000	30,000,000	14,000,000
Tracked Words / Unique Addresses	486 / 123	75,323 / 1,600	27,529 / 917	518 / 141	47,745 / 4,440
Largest Dataflow Words / Addresses	154 / 26	71,345 / 1,119	20,663 / 315	173 / 46	46,214 / 3,945
Number of Dataflows	10	29	50	10	46

Table 1. Experimental benchmarks. The table reports the number of execution cycles, tracked words and unique addresses, size of the largest dataflow and total number of dataflows for each benchmark.

ploits in the past. This benchmark is a simulated SQL injection attack where data from an HTTP request is used unsafely to generate a would-be SQL query. It returns the compromised query as an HTTP response.

3.3. Taint Analysis Coverage and Performance

We evaluated the performance impact of Testudo in a taint analysis application. The sample cache simulator was used to emulate the sampling of dataflows obtained from the data tracking simulator and to record the average number of runs required to achieve 99% dataflow coverage. We performed the analysis using a broad set of sample cache configurations: fully associative cache with 32, 64 and 128 entries, and 4-way set associative caches with 256, 512 and 1024 entries. The intra-flow eviction policy for the sample cache was uniformly random, while the inter-flow policy used an epoch length of 10 tainted dataflows and a 10% probability of winning at each challenge (see Section 2.2). The results of these experiments are shown in Figure 5. In each graph we plot the average coverage achieved for the total number of tainted words versus the number of runs required to achieve that coverage. Most benchmarks have been evaluated with more than one sample cache size (however, we show only a subset of our results due to space limitations). In all cases, we find that high coverage can be attained by Testudo with a reasonable number of runs, even with very limited cache sizes. As intuition suggests, increasing the size of the cache increases the coverage for a given benchmark, or reduces the number of runs required to reach a desired coverage target (*e.g.*, *telnet.server* in Figures 5.e and f). Once the cache size is sufficiently large to fully store the largest dataflow in a program, further increases do not provide any additional benefit (*e.g.*, *tiff.good* in Figures 5.b and c).

The access time for the sample caches used in our experiments is smaller than the CPU clock cycle and L1 data-cache access time of the systems that we emulated. Moreover, the propagation and checking of taint bits can be implemented efficiently within the pipeline, requiring no support from the policy map and the service handlers. Therefore, Testudo can perform taint analysis without incurring any performance penalty and without affecting program runtime. Thus, for this application, we provide a flexible trade-off between the sample cache size and the number of executions required to achieve high coverage.

3.4. Worst Case Analytical Bounds

To compare the actual performance of Testudo to the worst-case analytical model outlined in the appendix, we computed the analytical upper bound for a subset of the benchmarks listed above. Note that to conduct this analysis we must know the coverage probabilities for each tracked word in a benchmark (as many as 200,000 words for *xpdf.good*). The straightforward solution to computing the coverage probability for each word is based on a Markov-chain analysis of the dataflow. However, this approach becomes computationally intractable very quickly with growing dataflows and sample caches. To cope with this challenge we set up a Monte Carlo simulation with each fixed cache configuration and simulated our benchmarks until each tagged word in the program's dataflows was covered at least a few times. Each coverage probability (i.e., $p\_covered_i$ in the appendix) was then obtained by dividing the number of times each word was covered by the number of Monte Carlo runs.

Figure 6 shows the comparison between the experimental dataflow coverage of Section 3.3 and the analytical model derived in the appendix. The gray plots report the average coverage achieved after 80 trials versus the number of runs, while the black plots indicate the probability of achieving full coverage in a given number of runs using the analytical model. For example, full coverage of *tiff.bad* can be achieved with 80% probability after 400 runs when using a 64-entry sample cache. As Figure 6 demonstrates, Testudo can in practice achieve 99% coverage up to 5x times faster than the predicted worse-case number of executions. While the worst case number of executions to achieve 99% coverage may seem a conservative approximation, consider that Office XP sold sixty million licenses in its first year on the market [18]. Thus, assuming that each licensed user runs Microsoft Word twice per week, there are more than 12,000 executions every minute. Therefore, according to the worst case analysis model, achieving a 99% confidence of observing an event that occurs only once in ten thousand runs requires 46,000 runs, which would be gathered in less than 4 minutes at that rate. Apache, another popular program that runs on 82 million web servers as of April 2008 [20], runs 570,000 times per minute, assuming that each web site serves up to 10 pages per day. Clearly, these popular applications provide more than enough user executions to demonstrate the great benefit of a scalable approach to taint or security vulnerability analysis.

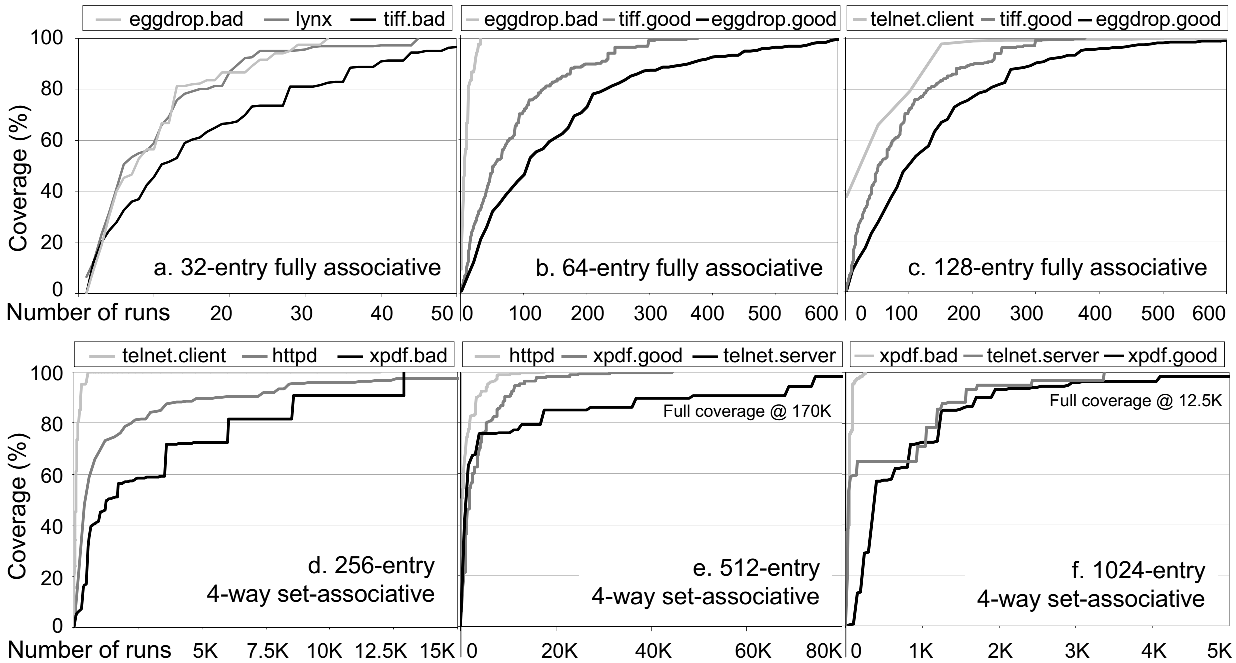


Figure 5: Dataflow coverage vs. program executions for various cache setups. The figures plot the number of runs required to achieve levels of dataflow analysis coverage. Increasing sample cache sizes reduce the runs required to achieve full coverage, up to a cache size where the largest dataflow can be fully stored (*telnet.server* in e and f). Beyond this threshold, no additional benefit is provided by larger caches (*tiff.good* in b and c).

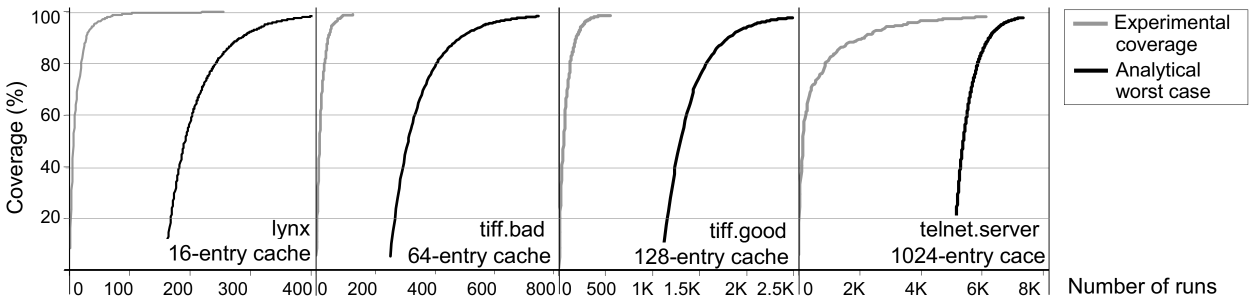


Figure 6: Comparison between experimental coverage and analytical upper bound. The plots report coverage vs. number of runs for both our experimental results (over 80 trials) and the analytically computed upper bounds (see appendix). The analytical upper bound is the probability of achieving full coverage after a particular number of runs. For instance, *tiff.bad* benchmark has an 80% probability of achieving full coverage in 400 runs. In practice, Testudo achieves 99% coverage in many fewer runs.

3.5. Beyond Taint Analysis

In this section we investigate performance overhead of conducting heavyweight security vulnerability analysis. While we showed in Section 3.3 that Testudo does not incur performance overhead when applied to taint analysis, this may not be the case for heavyweight vulnerability analyses, such as input bounds checking. In this case, the size of the tag associated with each tracked variable is more than a single bit and the analysis itself must be conducted in software by means of service handler routines, incurring an overall performance slowdown at each routine call. As a consequence, to retain acceptable overall performance, we must limit the number of tagged variables sampled by Testudo in each program execution (see Section 2.2). This down-sampling, in turn, increases the number of runs required to provide adequate coverage of the program's dataflows. To gauge the impact of heavyweight DIFT semantics with many service handlers, we plotted acceptable performance impact as a function of runs required to achieve 95% coverage of tagged words in Figure 7. The plots show the slowdown versus runs tradeoff for several handler overheads, ranging from 10 to 100 instructions executed for each call to a service handler. As a point of reference, the heavyweight input bounds analysis technique incurs approximately 50 instructions per instruction accessing a tagged value. Even with fairly intensive service handler routines, Testudo can maintain a tolerable to very small level of performance degradation perceived by the user by spreading the analysis over more executions.

3.6. Analysis of Sample Cache Overheads

In this section we analyze the performance, area and power requirements of our sample cache hardware. We designed the sample caches in our experiments using CACTI v5.0 [31] at 65nm/1.1V technology node. We evaluated fully associative caches with 16 to 128 entries, and 4-way set associative caches with 256 to 1024 entries. Each entry is five bytes

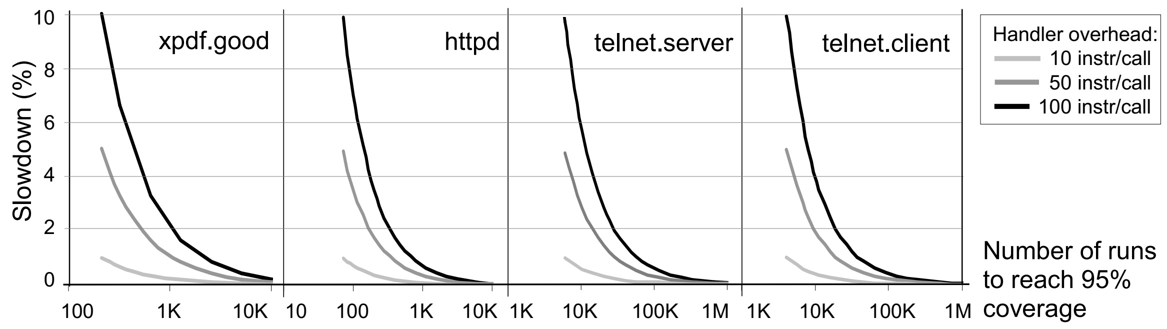


Figure 7: Performance overhead for heavyweight security analysis. As discussed in Section 2, heavyweight security analyses are carried out in Testudo through service handler routines linked in the policy map. The plots compare perceived execution slowdown vs. number of runs to achieve 95% coverage for three overhead levels incurred through the handler routines.

wide, a typical physical address width in modern processors [10]. We estimate the impact of including the sample caches into a complex modern processor, a 2.5GHz AMD Phenom [34], and into a simpler core (1.4GHz UltraSPARC T2 [19]).

Figure 8 plots access latency as a fraction of the CPU clock cycle, area and power overheads for the sample cache configurations we considered. The first seven bars in each graph compare the sample cache against the L1 cache of the AMD Phenom. In the Phenom processor, L1 cache accesses are pipelined and take three cycles to return. We thus show our cache statistics as if they were pipelined across multiple cycles. The final bar shows our largest cache size, 1024 entries, versus the UltraSPARC T2. The L1 caches in the T2 cores are much smaller than those in the Phenom, but they are also non-pipelined and accessed in a single cycle. Part a) of the figure shows the access time to the sample cache as a percentage of the respective processor's clock cycle. Note that in all cases, the access time to the sample cache is smaller than the L1 cache access time. Thus, no performance impact is experienced when the system is augmented with Testudo's sample cache hardware. We show in part b) that the area impact of the sample caches is minimal, less than 1% in all configurations. The graph compares the sample cache's area against the core and L1 cache area of the Phenom processor ($25.5mm^2$ as reported in [7]), and against the Niagara's core plus cache combination ($12.5mm^2$) in the last bar. Finally, part c) shows worst-case power requirements. To model a worst case scenario for the sample cache, we assumed that each instruction is a memory operation accessing the sample cache. As indicated by the figure, even worst-case power requirements are quite modest. In practice, one would expect these power requirements to be much less than this value since not all instructions access the sample cache, making the total power overhead very small. Note that the Phenom and UltraSPARC T2 dissipate 125W and 84W respectively.

We did not analyze the overhead incurred due to the pipeline modifications needed to include the tag bits in the register file and propagate them through the pipeline. It is unlikely that this hardware, which is operated in parallel to the rest of the pipeline, would affect the core's critical path.

4. Related Work

There have been many publications on both heavyweight security analysis systems and statistical bug-testing methods; our design takes inspiration from, and improves on, works from both of these areas. In this section, we summarize previous works and compare them against our technique.

4.1. Heavyweight Security Analysis Systems

We use the concept of *heavyweight analysis* as presented by Nethercote and Seward [22] to describe the types of analysis that Testudo can accelerate. Examples include taint analysis [23], secrecy analysis [17], and array bounds checking [21]. Testudo can significantly reduces the amount of slowdown that any individual user perceives while running heavyweight analysis by performing *distributed statistical* heavyweight analysis.

Taint analysis is an active area of research that focuses on using shadow values to track untrusted data from I/O and detect whether it is used in dangerous ways. Xu *et al.* [35] and LIFT [25] show how taint analysis systems can be used to detect many types of security vulnerabilities. The largest issue with taint analysis in software is the tremendous slowdown associated with it. Ho *et al.* [9] and Nightingale *et al.* [24] show methods for speeding up taint analysis in software. However, these and other software schemes still have slowdowns that are unacceptable for widespread deployment to end users.

Other researchers have worked on hardware methods to accelerate taint tracking. RIFLE [32], Minos [5], Suh *et al.* [28] and Chen *et al.* [2] all implemented hardware taint analysis schemes that significantly reduce the runtime overhead of taint tracking at the expense of increased system complexity and memory overheads. Because taintedness must propagate throughout the system (*e.g.*, from cache to memory), buses, caches, and memories must be expanded to accommodate the extra bits. With intelligent compression of taintedness information, memory overhead can be reduced from 12.5% to only a few percent in the average case. However, hardware designers may be wary of the potential memory overheads and increased design complexities of these schemes. Additionally, it is unlikely that these approaches could be extended to more

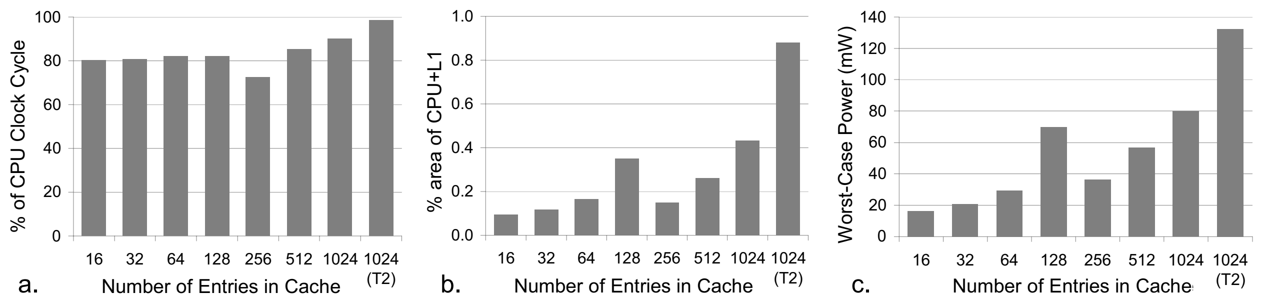


Figure 8: Delay, area and power overhead of sample cache designs. From left to right, the charts show access latency, area, and worst-case power estimates for several sizes and configurations of the sample cache. The first four bars in each graph are for fully-associative caches, while the last four correspond to four-way set associative configurations. The first seven bars in each graph are compared against a 2.5GHz AMD Phenom, while the last one is against a 1.4GHz Sun UltraSPARC T2.

heavyweight DIFT techniques without excessive overheads. Dalton *et al.* proposed Raksha [6] as a solution to correctness problems in existing hardware taint systems; it utilizes user-level exception handlers to allow the taint analysis rules and error-cases to be much more flexible. Similarly, flexibility is one of the primary goals of FlexiTaint [33], another taint system that focuses on making the hardware taint propagation as programmable as possible.

Bounds checking is another useful heavyweight analysis scheme. Nethercote and Fitzhardinge show how to use Valgrind to bounds-check data and check that it is safely used [21], but the overheads of this design are high (up to 80x). Alternately, Larson *et al.* also present a method for checking bounds on all inputs of a program [11] that has up to 200x runtime overhead. Lam and Chiueh show a software method for finding array bounds violations using features of the x86 instruction set [12]. This method has reasonable overhead, but it is limited in its power and flexibility.

The primary advantage of Testudo over other hardware-based analysis systems is its ability to do distributed heavyweight analysis with little extra hardware and minimal runtime overhead. Given these capabilities, Testudo makes it possible to widely distribute security vulnerability analysis infrastructure to end users, with the promise of providing unprecedented levels of analysis coverage.

4.2. Sampling Systems

Besides heavyweight analysis, there have been a number of works on statistical sampling systems for software debugging. Liblit *et al.* show a number of methods for using statistical sampling to find bugs in software based on the sampled runs of thousands of users [13, 14]. The idea of sampling over the user population in order to find problems while maintaining low runtime overhead is similar to what we propose, though the goals of the two systems are different. Liblit's system performs data mining of dynamic invariants and violations of those invariants that indicate a possible bug. Testudo, on the other hand, performs distributed dataflow analysis for the purpose of locating security vulnerabilities. In addition, Testudo employs a novel feedback-controlled random selection policy which allows for the low-cost distributed analysis of much more compute-intensive analysis schemes. Chilimbi and Hauswirth implemented a sampling system for finding memory leaks in software [3]. Their system finds objects that have been leaked by an adaptive program profiling scheme. This results in low overhead, but the requires a great deal of binary information.

5. Conclusions

The challenge of excising bugs from software applications is quite critical, especially when considering that most software security vulnerabilities exploit software bugs to implement attacks. To protect programs, dynamic information flow tracking (DIFT) techniques have emerged as a powerful solution that identifies improperly used external inputs in programs. Unfortunately, the high performance overheads of DIFT techniques have limited their use to the research laboratory. If very low-cost techniques could be deployed widely to users, it would be possible to achieve levels of security vulnerability analysis previously not possible.

To this end, we have proposed a hardware-based programmable DIFT analysis technique called *Testudo*. The approach is based on a dynamic distributed debugging technique that spreads the analysis workload over many executions of the program. A random selection mechanism picks external inputs on occasion and follows their dataflow executions, looking for dangerous uses of externally derived variables. We can keep memory overheads low through the use of a sample cache that holds the value tags of a few tracked variables. The sample cache utilizes a random replacement strategy that ensures that successive runs of the program are likely to see different externally derived variables. Thus, multiple executions of the program analyze additional program computation. To minimize the performance impact, the random selection mechanism dynamically monitors performance overheads and limits the introduction of new dataflows into the sample cache if overheads encroach the budgeted performance slowdown.

To understand the nature of this approach, we analyzed its performance empirically with full-system simulation as well as by developing an analytical model that estimates the number of program runs necessary to achieve high-coverage analysis of the program. We find in our experimental results that our approach is quite low cost, and it achieves high program coverage often with a mere thousands of executions in the worst case. For popular programs that are the most frequent target of security attacks, their many users would provide a level of security vulnerability analysis that is simply not possible with solutions available today. Additionally, we examined latency, area, and power costs of the sample cache hardware and found that sample caches of up to 1024 entries provide exceptionally good coverage and only increase system costs by a few percentage.

Looking forward, there are a number of improvements possible to our initial design. First, we are working to refine the sample cache replacement policy to further reduce the number of executions required to analyze large dataflows in the worst case. Second, we are working to implement additional security vulnerability analysis techniques, in particular fully-symbolic input bounds analysis and dynamic type checking. The flexibility of our DIFT analysis framework should accommodate these new analyses with little or no hardware changes. Finally, we are working on extending the sample cache to provide caching benefits to traditional non-sampled DIFT analyses. These extensions will allow traditional non-sampled software-only DIFT techniques – that developers currently run in the development laboratory – to execute faster because tag values can be more quickly accessed in the sample cache.

References

[1] Bochs: The cross platform IA-32 emulator. http://bochs.sourceforge.net.

[2] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of Dependable Systems and Networks (DSN)*, 2005.

[3] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[4] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. Insecure programming: how culpable is a language's syntax? In *Proc. of Workshop on Information Assurance*, 2003.

[5] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of International Symposium on Microarchitecture (MICRO)*, 2004.

[6] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc of International Symposium on Computer Architecture*, 2007.

[7] H. de Vries. Chip-Architect.com, Feb. 2007. http://www.chip-architect.com/ news/2007_02_19_Various_Images.html.

[8] C. Evans. Libtiff-3.6.1 image decoder parsing flaws. http://scary.beasts.org/security/CESA-2004-006.txt, 2004.

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

[10] Intel Corporation. Intel 64 and IA-32 software developer's manual - volume 3A. http://developer.intel.com/design/processor/manuals/253668.pdf.

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

[12] L.C. Lam and T.C. Chiueh. Checking array bound violation using segmentation hardware. In *Proc. of Dependable Systems and Networks (DSN)*, 2005.

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

[14] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. of ACM Conference on Programming Language Design and Implementation*, 2005.

[15] M. Bushnell, V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI circuits*. Springer, 2000.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2), 2002.

[17] S. McCamant and M. D. Ernst. Quantitative information-flow tracking for c and related languages. Technical report, MIT-CSAIL, Nov. 2006.

[18] Microsoft Corp. Microsoft 2002 annual report and form 10-k. http://www.microsoft.com/msft/ar02/.

[19] U. G. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC: (Niagara 2). In *Proc. of International Solid State Circuits Conference (ISSCC)*, 2007.

[20] Netcraft Ltd. April 2008 web server survey. http://news.netcraft.com/archives/2008/04/14/april_2008_web_server_survey.html.

[21] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proc. of Workshop on Semantics, Program analysis, and Computing Environments for Memory Mamagement (SPACE)*, Jan. 2004.

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

[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2005.

[24] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[25] 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 *Proc. of International Symposium on Microarchitecture (MICRO)*, 2006.

[26] SecurityFocus. Lynx NNTP article header buffer overflow vulnerability. http://www.securityfocus.com/bid/15117/info, Nov. 2005.

[27] SecurityFocus. Eggdrop server module message handling remote buffer overflow vulnerability. http://www.securityfocus.com/bid/24070/info, May 2007.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[29] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

[30] The Month of Apple Bugs. Multiple vendor PDF document catalog handling vulnerability. http://projects.info-pull.com/moab/MOAB-06-01-2007.html, June 2007.

[31] S. Thoziyoor, N. Muralimanohar, and N. Jouppi. CACTI 5.0. http://www.hpl.hp.com/techreports/2007/HPL-2007-167.pdf, Oct. 2007.

[32] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. of International Symposium on Microarchitecture (MICRO)*, 2004.

[33] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *Proc. of International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[34] B. Waldecker. AMD Quad Core Processor Overview. http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/AMD_ORNL_073007.pdf.

[35] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practial approach to defeat a wide range of attacks. In *Proceedings of USENIX Security Conference*, 2006.

Appendix: An Analytical Model of Coverage

Dataflow sampling is a decentralized statistical approach to security vulnerability analysis. The small storage available in the sample cache limits the likely coverage achieved during a single program execution to only a few externally derived dataflows, and only a portion of the larger dataflows. Since the replacement policy encourages covering different dataflows (and portions of dataflows) in subsequent executions, the analysis will eventually achieve full coverage. Here we derive an analytical model that delineates an upper bound for the number of executions required to achieve full coverage. The model is parameterized with the size of the sample cache and the frequency at which tagged words are considered and analyzed. This frequency is a function of the number of dataflows in the program, the dataflows' shapes and the specific cache replacement policies selected in the analysis (see Section 2.2). The analytical model that we derive in this appendix is an upper bound on the number of executions required to achieve full coverage with a given probability – that is, to cover a user-specified fraction of all the tagged words in a program. Section 3.4 compares the upper bound derived with this analytical model against actual empirical measurements gathered in our experimental evaluation and shows that we perform much better in practice.

Overview of the Analytical Model. Our analytical model computes the probability that a particular dataflow word has not been covered after a program execution. Using probability theory concepts, it is then possible to compute the probability that there exists *any* word in *all* dataflows that has not yet been covered after a program execution. We finally use this expression to extend the model to estimate the probability of having an uncovered tagged word after N executions. This probability can be plotted as a function of N, such that it is possible to see the probability that full coverage is not reached after a given number of runs (see Figure 6).

Derivation of the Analytical Model. The first step of the model requires characterization of the program's dataflows to compute the probability p_cov_i that any given word i resides in the sample cache configuration at the end of a program execution or has been removed from it only after the word's last use in the program. While this value could be computed analytically for small sample cache configurations by fully enumerating all possible caching scenarios, it is computationally infeasible to perform this step of the analysis analytically for larger sample caches. Consequently, we measure these probabilities empirically by running Monte Carlo simulations on the externally derived dataflows of a program for a given sample cache configuration. Once we reach the desired confidence level, the p_cov_i are computed as the fraction of times each tagged word is covered during the total number of simulated executions.

Given the probability p_cov_i of an individual word being covered in one program execution, we can compute the probability that this word has *not* yet been covered after N runs as $(1 - p\_cov_i)^N$. From analytical calculus, this latter expression is always less than or equal to $e^{-p\_cov_i \cdot N}$, for p_cov_i between 0 and 1. Hence, we can use the exponential as a conservative approximation of the original expression. That is, the probability that the tagged word i has *not* been covered after N runs is always less than or equal to $e^{-p\_cov_i \cdot N}$.

Finally, we extend this inequality to include *any* tagged word in *all* dataflows by using the *union bound*: the probability that any tagged word in all dataflows has not yet been seen after N program executions is less than or equal to the sum of the individual events' probabilities. Thus, *the probability ϵ of having any uncovered tagged word after N runs is:*

$$\epsilon = \sum_i e^{-p\_cov_i \cdot N}$$

Thus, given the individual probability of covering each tagged node i, we can compute the upper bound probability of having any uncovered tagged word after N runs. By plotting this result for increasing values of N, we can find the first value of N for which ϵ is less than $1 - desired\_confidence\_level$. For example, if we want a 90% confidence of covering all tagged words in a program, we sweep over N until we derive an $\epsilon \leq 0.1$. Note that the union bound provides a very good approximation if the events are independent; when the events are correlated, the approximation becomes coarser, but is still conservative.

It is important to note that the model computes a worst-case estimate to achieve the desired confidence. As shown in Section 3.4, the empirical results often require fewer executions to accomplish the same goal. Any significant difference between the analytical model and the experiments is due to dependencies between the events (that is, covering tagged word i correlates with covering word j). The more dependent the events, the fewer execution will be required in practice to achieve full coverage. Nonetheless, the result of the analytical model is valuable because it equips developers with an approximation of how many executions will be required, in the worst case, to achieve full program coverage.