Vulnerability-Specific Execution Filtering with Log-Based Architecture Lifeguards

15-740 Final Report

Peter Chapman
peter@cmu.edu

Deby Katz
dskatz@cs.cmu.edu

Stefan Muller
smuller@cs.cmu.edu

December 4, 2012

All project documents are available at `http://www.cs.cmu.edu/afs/cs/user/pmchapma/www/15740.html` .

1 Introduction

Instruction-grain dynamic monitoring tools can detect bugs and prevent security violations in executing programs. Traditionally, instructions from the monitoring tool are inserted into the currently executing program using well-established techniques such as binary rewriting, software-based emulation, or binary instrumentation in order to provide timely detection and possibly mitigation. Those techniques not only interrupt and temporarily halt the program execution, but also disrupt the resource allocation of the application (e.g., evicting register values to hold monitoring information). The Log-Based Architecture (LBA) [3] has been proposed as a method to utilize extra hardware resources (i.e., extra processing cores) in order to provide the expensive quality assurance properties of dynamic monitoring tools with only minimal slowdown.

The support and optimizations of LBA can reduce typical monitoring overhead by an order of magnitude or greater. However, the monitoring of non-trivial properties on LBA can still slow application execution by 2-3×. In this work, we apply vulnerability-specific execution filtering [10] to instrument only instructions critical to specific application vulnerabilities and reduce the overall slowdown to less than 25% for typical executions. Additionally we perform a brief comparison of LBA to the state-of-the-art software-based taint tracking tool, Minemu [1]. Lastly, we discuss the possibilities presented in future work by the minimal overheads imposed by this configuration.

2 Related Work

Dynamic Monitoring Researchers have developed numerous dynamic monitoring software tools that watch events related to program execution in order to detect bugs [8, 13] and security violations [1, 4, 5, 6, 7, 8, 11, 12]. While many tools focus on high-level events such as system calls [7, 11], there is a wealth of powerful monitoring tools and frameworks that rely on instruction-grain information [1, 4, 6, 5, 8, 12].

Taint Tracking In particular, the research community has heavily investigated taint tracking [14]. Taint tracking systems monitor the propagation of untrusted inputs through the execution of a program asserting that the inputs are not used unsafely (e.g., as the return address in a buffer overflow attack). Unfortunately the performance cost of instrumenting every instruction has historically prohibited taint tracking tools because they are impractical for real-world deployment. Specific hardware additions have been proposed [3, 15], but most work focuses on software-based optimizations [1, 4, 6, 5, 12] to varying levels of success based on different application [10] and hardware assumptions such as 64-bit processing cores [12] or SSE4 support [1].

Minemu Minemu [1] is the self-proclaimed "World's Fasted Taint Tracker", reporting slowdowns less than 3× for real-world applications. The system achieves the speed-up primarily through two optimizations. First, Minemu stores taint tags in the extra hardware registers provided by the SSE4 x86 extensions to avoid evicting application data. Second, taint tags are stored in a single-level shadow table to minimize propagation and assertion logic. These optimizations limit deployment of Minemu to recently built systems with significant main memory. Regardless, Minemu can still be practically deployed on many commodity systems and so we compare the performance of this primarily software-based taint-tracking approach to the general-purpose hardware enabled optimizations in LBA (see §5.3).

LBA The Log-Based Architecture [3] is a proposed set of hardware additions that leverages spare cores in a multiprocessor system to decrease the cost of dynamic execution monitoring. The decreased overhead is achieved by assigning one processor core the role of monitoring the execution of an application on a separate core with instruction granularity. LBA advocates adding specialized hardware to log instructions executed on the application core and send to another (otherwise inactive) core, where the instructions can be analyzed without interfering with the execution of the primary application. Specifically, this analysis does not compete for cycles, registers, or the L1 cache with the running program. LBA proposals have included several "lifeguards" (programs running on the monitoring core), one of which is the taint tracker TaintCheck [3] . Taint tracking using LBA has the potential to be efficient enough for deployment systems.

VSEF Vulnerability-specific execution filtering [10] (VSEF) was proposed as a key component in a worm defense system [2]. The primary focus of the VSEF technique is to quickly defend against specific threats with little overhead. To that end, the prototype VSEF implementation modified Valgrind [8] to only instrument instructions present in a vulnerability-specific filter with the taint-tracking module TaintCheck. The filter is a list of the instructions along the exploitative path in the data-flow graph created using a malware sample. The prototype incurred less than 3% overhead beyond the cost of the Valgrind instrumentation, a slowdown around 3-10× for typical programs. To overcome this limitation the authors proposed re-instrumenting the vulnerable binary to include the taint-tracking, but this system was never built. In this work, we prototype a hardware filter addition to LBA that combines the precision of VSEF with the performance gains of LBA.

3 Filter Generation

Ideally, we would frame the filter generation process as a black-box input to our system. As no public implementation for VSEF-generation exists, we developed our own manual process, but if

necessary, the entire process could be automated in the manner described by prior work [2]. Given a target binary executable and a malicious input we record an execution trace using the Binary Analysis Platform[1] (BAP). With the trace, we use BAP to create a data flow graph of the trace from which we can extract the set of instructions that must be allowed through the filter to detect exploits against the vulnerability. The primary limitations of this process are the space needed to record the execution trace and the time needed to generate the data-flow graph. For long execution traces a purpose-built tool for creating filters may be necessary.

4 Simulated Hardware Filtering

Filters that are generated as described can be integrated into the LBA architecture. Using LBA, instructions can be filtered at several levels. It would be possible to implement filtering at the software level by adding an additional layer between the hardware and the lifeguard. This layer would invoke lifeguard instruction handlers only if the instruction being processed is present in the filter. The filter would be stored in a designated file in the file system. However, such an implementation would need to check every instruction executed in the application against the filter. Even if the filter were stored entirely in the cache, this process could be a very expensive operation costing tens of cycles or more per application instruction.

Instead, we implement the filter in the hardware of the LBA architecture. This allows the vulnerability-specific filter to, for example, be loaded into a designated cache, with specialized hardware for fast lookup. When the dispatch hardware fetches a record, instead of immediately executing the lifeguard handler, it first looks up the memory address of the executed instruction in the filter. If the address is not present in the filter, the lifeguard is not invoked and the next record is fetched immediately.

To evaluate this modification to the architecture, we have modified the existing LBA extension of the Simics simulator to simulate hardware filtering. When the LBA system initializes, the filter, in the form of a list of memory addresses, is loaded from a designated file into memory. Addresses are stored using the `set` class from the `C++` Standard Template Library. The code for fetching log records is extended with code to check the instruction address against the filter. If the instruction address is not present in the filter, no further processing is done on this record and the next record is fetched immediately. Because this code is in the Simics extension, it should not be included in the measured execution time of applications run in the simulator, reflecting our assumption that the filtering operations can be implemented efficiently in hardware.

5 Evaluation

We evaluate the performance of our prototype implementation of VSEF on LBA in three ways. First, we discuss the security guarantees provided by a filter (§5.1). Second, we examine the performance gains of applying a filter to a real-world program in typical scenarios (§5.2). Third, we evaluate the performance of our system, which requires non-standard hardware, to the state-of-the-art taint tracking system built to use commodity hardware, Minemu (§5.3).

[1] `http://bap.ece.cmu.edu`

Benchmarks Table 1 gives a brief description of all the benchmarks used in our experiments. The three applications, Ghostscript, `gzip`, and `tidy`, were chosen due to the nature on which their execution is highly dependent on the input (i.e., postscript interpretation, compression, and HTML processing). Our goal was to present realistic, yet difficult, applications and inputs. For experiments that require a filter, the first item listed for each application (gs-isca96, gzip-readme, and tidy-readme) was used as the "malicious'' input to generate the filter. We treated these inputs as undesired in our experiments because we wanted to test how complex inputs and instrumentation would affect filter performance. We felt that an experiment based on filtering a small exploit such as processing command-line arguments would not be informative of the performance of VSEF on LBA. Unfortunately we could not find sufficiently complex real-world exploits, and since our primary goal was to evaluate the performance of the system, we opted to use what would otherwise be benign inputs.

5.1 Security Properties

The TaintCheck lifeguard used in LBA has the same security properties as the source algorithm [9]. A thorough discussion of the accuracy of TaintCheck requires a precise definition of permitted and prohibited executions. For example, a buggy application can allow unsafe inputs to manipulate the control flow of an application in a unsafe manner, but few taint tracking systems propagate taint as the result of control flow changes in order to avoid over-tainting. Whether this is a false negative depends on the definition of a permitted execution. For our purposes we want to focus on the security properties of VSEF.

For identical inputs, VSEF does not affect the accuracy of TaintCheck as all the critical instructions will be sent to the lifeguard for analysis. Furthermore, as long as the execution accesses the same sequence of basic code blocks, accuracy is unchanged from TaintCheck. This stipulation includes different loop iteration counts; a useful property for vulnerabilities whose filter should handle different length inputs. However, if the trace deviates from the code used to generate the filter, false negatives may occur. The TaintCheck module only sees instructions white-listed by the filter. From the perspective of the taint tracking algorithm, an execution can begin with tainted data, follow an execution path that sanitizes the input, return to the path for the filter and cause a false positive security violation. The same fundamental limitation can also cause false negative results when unsafe input propagates outside of the monitored path. As noted by the original work [10] a runtime memory monitor can eliminate the false positives but at a great cost to performance. The false negatives cannot be mitigated without complete monitoring of every execution as the filter does not provide enough context for all possible execution paths.

5.2 Filtering Performance

We conducted tests within the simulated LBA system to evaluate the performance of our filtered-TaintCheck modifications to LBA. We used two baselines for these tests. The first was running each benchmark and input within the LBA simulator but without any lifeguard processing. These tests established the baseline for performance of each process within the simulator. The second baseline was running each combination of benchmark and input within the LBA simulator with the full TaintCheck lifeguard running. These tests provided the baseline for how much the addition of a filter improved performance.

Benchmark	Program	Input
gs-isca96	Ghostscript	LBA research paper (309K)
gs-doretree	Ghostscript	A 3D graphic of a tree (137K)
gs-milestone	Ghostscript	15-740 project milestone report (242K)
gzip-readme	`gzip`	A small readme file (10K)
gzip-psfiles	`gzip`	Two PostScript files (465K)
gzip-tcm	`gzip`	JPEG image of Prof. Todd Mowry (166K)
tidy-readme	`tidy`	A small readme file (no HTML, 174b)
tidy-make	`tidy`	GNU make manual (811K)
tidy-fc2	`tidy`	Fedora Core 2 introduction (70K)

Table 1: Benchmark Descriptions

When we compared the experiments conducted without any lifeguard to the experiments conducted with the full TaintCheck lifeguard running, we observed that the TaintCheck lifeguard slowed the execution of the benchmark programs by 2 to 4.2×. These numbers are consistent with those observed in prior work [3]. By contrast, when TaintCheck was run with a filter, the executions were only slowed by 1.04 to 1.91× compared to the baseline simulated runs without a lifeguard. When we compared the filtered executions to the full TaintCheck executions, the filtered executions ran between 1.07 and 4.03× faster, as shown in Figure 1. The reduction in time between the full TaintCheck tests and the filtered TaintCheck tests reflects that, in the filtered setup, the lifeguard needs to process fewer instructions.

One interesting result occurred in the gs-isca96 benchmark. When run with the filter generated from that input, the performance improved by 2.84× compared to the full TaintCheck. This result was unlike those obtained for the other benchmarks when using a filter generated from the corresponding input – their speedups were much smaller at 1.07 and 1.51. A smaller speedup in this case could be expected because every instruction address listed in the filter must be executed at some point during the run of the benchmark and is therefore passed to TaintCheck. Nonetheless, this case points out that performance overhead with hardware-based VSEF is a function of the number of instructions passed to TaintCheck, not the number of different instruction addresses. In this benchmark the portion of the execution that involved unsafe data manipulation was small enough that the number of instructions filtered out improved performance significantly.

We conducted all tests within the simulator using a consistent configuration for each test. As discussed in the introduction to this section, it should be noted that we did not use filters that came from real-world exploits and so typical performance in a deployed system could vary from the measurements taken here.

5.3 Taint-Tracking Comparison

We compared the performance of the VSEF-filtered TaintCheck lifeguard running on LBA with Minemu [1], a fast software-based taint tracker that runs on commodity hardware. The benchmarks described in §5.2 were run on a virtual machine running Ubuntu 8.04, natively and with Minemu. Minemu could not be run on the LBA simulator because of its hardware requirements. However, we believe we still achieve meaningful results because we report normalized execution time. The LBA TaintCheck performance is compared to the performance of the benchmark with no lifeguard, as in §5.2, and the performance of the benchmark under Minemu is compared to the performance

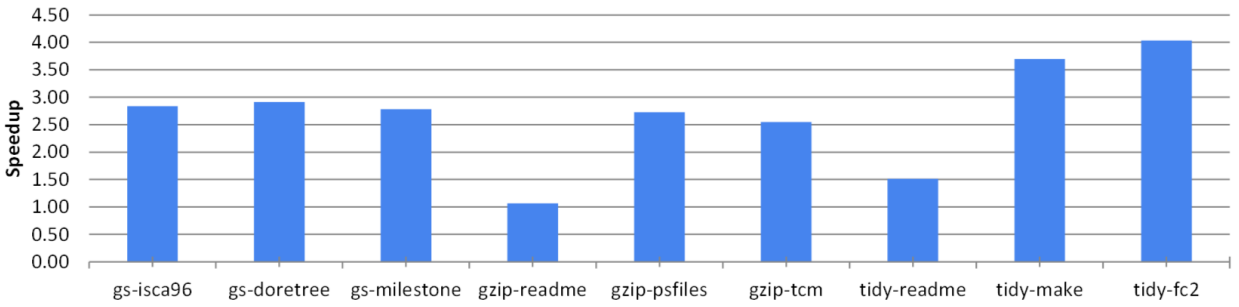

Figure 1: Speedup from VSEF (higher bars are better)

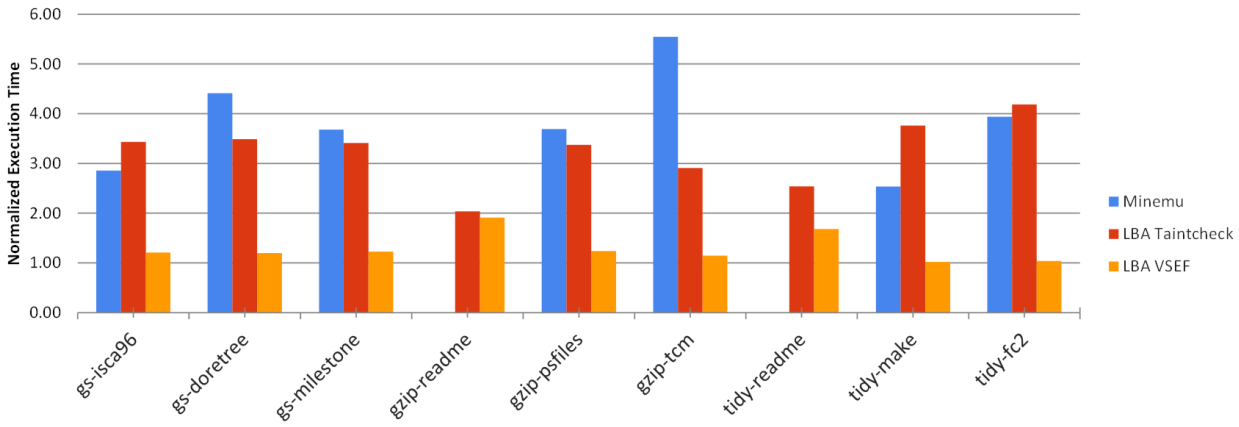

Figure 2: Normal Execution Time (lower bars are better)

of a native run of the benchmark on the same system.

Table 2 shows the execution times for each benchmark under Minemu, full LBA TaintCheck and LBA TaintCheck with VSEF, all normalized appropriately as described above. The last column shows the speedup achieved by using VSEF over full TaintCheck (this is the third column divided by the fourth column.) These results are also shown in Figure 2.

Minemu shows an unusually high slowdown for gzip-readme and tidy-readme because the very small size of these benchmarks caused the overhead inherent in Minemu to dominate execution time. For clarity, these outliers are not shown in Figure 2. Otherwise, the results achieved for Minemu (slowdowns of approximately 3×) are consistent with those reported by its developers. While the performance of Minemu is comparable to that of unfiltered LBA on most benchmarks, our approach is consistently faster than both other methods tested.

6 Discussion

The experimental results show that VSEF is a promising approach for achieving security properties comparable to those of TaintCheck with lower overhead. Because Minemu was not run in the LBA

	Slowdown			
Benchmark	Minemu	LBA Taintcheck	LBA VSEF	VSEF Speedup
gs-isca96	2.86	3.43	1.21	2.84
gs-doretree	4.41	3.49	1.20	2.91
gs-milestone	3.68	3.41	1.23	2.78
gzip-readme	32.47*	2.03	1.91	1.07
gzip-psfiles	3.69	3.37	1.24	2.72
gzip-tcm	5.55	2.91	1.14	2.55
tidy-readme	29.51*	2.54	1.68	1.51
tidy-make	2.53	3.76	1.02	3.69
tidy-fc2	3.94	4.19	1.04	4.03

Table 2: Experimental Results

simulator, the natures of the measurements used in the comparison differ, though we believe the comparison is still valid. In addition, the simulation does not take into account the overhead of filtering operations performed in hardware. However, by optimizing hardware for the simple tasks involved, these operations could be made quite efficient.

Completion of the Project

There were a number of surprises encountered during the project, most of them relating to the LBA platform. The code and simulator proved more difficult to set up than we anticipated, causing the project to fall behind schedule. The nature of the simulator, for example which operations are considered to occur in software and which occur in simulated hardware, only became clear to us after spending a substantial amount of time on the project and numerous discussions with some of the current members of the LBA project team. In retrospect, we wish we had more thoroughly explored the codebase and had the discussions before starting the project to gain a more thorough understanding and a better estimate of the timeframe.

Work on the project was split fairly evenly. Deby was the primary contact with the LBA developers and was responsible for setting up the environment in which to run our simulations. Peter worked on developing filters from binaries and on devising appropriate comparisons between our work and other taint-tracking approaches. Stefan modified the LBA Simics extension to perform filtering, and modified the Simics checkpoints and scripts in order to run the VSEF simulations. The milestone and final reports were produced collaboratively, with work split evenly among the group members.

Future Work

We have identified several promising areas for future work. It would be useful to evaluate a version of LBA TaintCheck with the VSEF filter implemented in software rather than in hardware. A comparison of software-implemented filtering to hardware-implemented filtering would confirm or refute our intuition that hardware-accelerated filtering is useful for keeping runtimes low. Another avenue of further research would be to experiment with ways to use computational power on the lifeguard that is not used by the VSEF-filtered TaintCheck lifeguard. Possible areas of investigation include adding increasing numbers of filters to the system. An investigation of the expected

additional overhead for an added exploit filter would provide insight into whether this would be a feasible use of the unused processing power. An additional area of investigation would be whether it is feasible to run an additional lifeguard simultaneously with filtered-TaintCheck to monitor a different property, such as full address checking.

7 Conclusion

In this work we applied hardware-based vulnerability-specific execution filtering to the Log-Based Architecture. We found that for typical executions the overhead was under 25%, making this technique an acceptable way to increase the security of vulnerable software in deployment given the hardware support. We additionally evaluated the performance of the state-of-the-art taint tracking software against the general purpose LBA and found that the performance to be comparable. However, it should be noted that each system has drastically different requirements and applications; neither system dominates the other. Lastly we discussed a few avenues for further research into blending VSEF and LBA.

Acknowledgements

We would like to thank Michelle Goodstein and Olatunji Ruwase for their generous assistance in understanding and running the LBA platform. We are also grateful to Professor Mowry and the 15-740 TAs for their time and dedication throughout this project and the course. This project funded in part by NSF Award 1065112 and this material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 0946825.

References

[1] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. *Recent Advances in Intrusion Detection*, pages 1–20, 2011.

[2] David Brumley, James Newsome, and Dawn Song. Sting: An end-to-end self-healing system for defending against internet worms. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 147–170. Springer US, 2007.

[3] Shimin Chen, Evangelos Vlachos, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, and Michael Ryan. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. *ACM SIGARCH Computer Architecture News*, 36(3):377–388, June 2008.

[4] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[5] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacymonitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.

[6] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L Fowler, and Murphy McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[7] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[8] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89:44–66, 2003.

[9] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[10] James Newsome, David Brumley, and Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. *Proceedings of the 13th Symposium on Network and Distributed System Security*, 9(May):1231–1242, 2006.

[11] Niels Provos. Improving host security with system call policies. In *In Proceedings of the 12th Usenix Security Symposium*, pages 257–272, 2002.

[12] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT : A Low-Overhead Practical Information Flow Tracking System for. *IEEE/ACM International Symposium on Microarchitecture*, 39, 2006.

[13] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[14] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317 –331, may 2010.

[15] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGPLAN Not.*, 39(11):85–96, October 2004.