Binary-level Function Profiling for Intrusion Detection and Smart Error Virtualization

Michael E. Locasto
Columbia University
locasto@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

Most current approaches to self-healing software (SHS) suffer from semantic incorrectness of the response mechanism. To support SHS, we propose Smart Error Virtualization (SEV), which treats functions as transactions but provides a way to guide the program state and remediation to be a more correct value than previous work.

We perform runtime binary-level profiling on unmodified applications to learn both good return values and error return values (produced when the program encounters "bad" input). The goal is to "learn from mistakes" by converting malicious input to the program's notion of "bad" input.

We introduce two implementations of this system that support three major uses: function profiling for regression testing, function profiling for host-based anomaly detection (envinronment-specialized fault detection), and function profiling for automatic attack remediation via SEV. Our systems do not require access to the source code of the application to enact a fix. Finally, this paper is, in part, a critical examination of error virtualization in order to shed light on how to approach semantic correctness.

1 Introduction

A key problem in computer security is the inability of systems to automatically protect themselves from attack. It is unlikely that system security problems will ever completely disappear. New and creative exploits will emerge to take advantage of mistakes in system design, construction, configuration, and deployment. Since it is difficult, if not impossible, to perceive or predict all threats *a priori*, no system can be completely secure. Furthermore, most exploits are launched with little or no warning, and the window to protect systems against *known* vulnerabilities is shrinking. Symantec reported [14] that the period of time from the announcement of a vulnerability to the appearance of an exploit was about 5.8 days in the first half of 2004. Current estimates are much shorter, and the recent Zotob worm was released 3 days after the vulnerability was announced. In the case of worms, malware spreads so quickly as to defy meaningful human intervention. In order to have a reasonable chance at surviving or deflecting an attack, a system must incorporate automatic reaction mechanisms. Such reaction mechanisms must be driven by intrusion detection sensors that perform unsupervised learning.

Recent advances in secure systems have led to an emerging interest in self-healing software as a solution to this problem. In many environments, continued execution is valued more highly than absolute correctness of the computation. For these types of environments, researchers have proposed two software self-healing mechanisms: *error virtualization* [11] and *failure-oblivious computing* [10]. While both of these approaches showed promising results as far as supporting system survivability and preventing the exploitation of vulnerabilities, they both suffer from the potential for semantically incorrect execution following the invocation of the self-healing mechanism. Such a shortcoming is devastating for applications that perform detailed and precise calcuations (e.g., scientific or financial software) or provide authentication or authorization services. In addition, the heuristics for error virtualization fail about 12% of the time for the applications that were tested. On applications that require continued availability, this failure rate is quite discouraging.

1.1 Error Virtualization

The key assumption underlying error virtualization is that a mapping can be created between the set of errors that could occur during a program's execution and the limited set of errors that are explicitly handled by the program code. By virtualizing the errors, an application can continue execution through a fault or exploited vulnerability by nullifying the effects of such a fault or exploit and using a manufactured return value for the function where the fault occurred.

The programs tested in [11] were network server-type applications – applications that typically have a primary request processing loop and can presumably tolerate minor errors in one particular trace of the loop. Furthermore, the system (STEM) proposed in [11] requires access to the source code of the application. These shortcomings motivate the development of a tool that can operate on an unmodified binary version of the application and learn appropriate "error virtualization" values during runtime (as opposed to statically determined heuristic values). In this paper, we describe the design, implementation, and evaluation of such a system. We also consider its application to regression testing and host-based anomaly detection.

1.2 Contributions

This paper makes the following four contributions:

- a model for binary-level function behavior profiling, including a spectrum for evaluating the correctness of a self-healing mechanism
- an implementation of this model as both a Valgrind [6] tool (Lugrind) and as a PIN [5] tool
- the notion of Smart Error Virtualization (SEV), a technique for self-healing that involves learning appropriate function return values at runtime to assist the basic error virtualization technique
- a study and critique of the efficacy of S/EV for a wide variety of applications

We stress that our system does not require access to source code or even a recompilation or restart of the system. Our primary goal is to support self-healing software and automatic error recovery with minimal or no downtime for the system as it recovers.

2 Approach

The design of a binary-level function profiler for SEV is driven by the hypothesis that "bad" or malformed input data and environment perturbations will drive function arguments and return values towards a profile of "failure dynamics", or operation under stress that manifests safe failure execution. Therefore, positive confirmation of this hypothesis improves on error virtualization because the application is trained on malformed or malicious input and the replacement values are selected from the previously observed training behavior. The design of our tools is aimed at profiling the execution of the application under this type of input and observing the values involved in successfully handling such input. If other input (such as that exploiting a vulnerbility) is supplied to the system, our tool can, upon detection of the exploit (via any number of techniques like taint-tracking, ISR, Dyboc, stack instrumentation, Valgrind's memory-checking, or OS-level exceptions), replace the state of the current function with previously learned state from the training model. While this reactive approach to self-healing is in keeping with previous work, we can also employ a pre-emptive approach, such as triggering the system with an anomally-based intrusion detection sensor, or even have a hybrid system.

2.1 Use Cases

We examine this hypothesis in a range of application and consider several ways to utilize the learned profile.

1. profiling of function call stack, return values, and parameter values for **regression testing**
2. detection of deviations from a learned profile for **host-based anomaly detection**
3. **software self-healing** through Smart Error Virtualization (S/EV) by picking an appropriate runtime-learned error code for virtualizing errors and slicing off a corrupted function.

One thing we can do is show how badly error virtualization behaves for certain classes of applications: like returning the same amount of memory for malloc (rather return NULL, how do we actually decide or learn this from runtime behavior). In particular, it would be nice (but time consuming) if the programmer could tell us what the possible set of error values for each function were through annotation. Barring that, we have to train on the regular "bad" execution of the binary. Our tool provides for the programmer specifying such a file, with no need to touch the source code or recompile. It is simply the model that is loaded in. Thus, this can be done by the system administrator or the programmer.

2.2 Recovery: Smart Error Virtualization

Previous work on error virtualization determined the return value of a sliced-off function by employing simple heuristics based on the return type of the offending function as determined by source code analysis. SEV attempts to discover more appropriate return values by keeping a record of the return values that have been seen in the program so far for various invocations of each function. These return values can also be selected using a conditional probability based on a matching of the call stack configuration at the time of use with similar configurations encountered during the training phase.

2.3 Range of Correctness

In order to have a reasonable basis for evaluating the basic error virtualization and SEV approaches to self-healing software, we need to develop a metric of system correctness. There are two types of correctness: the first with respect to continued execution and the second with respect to proper execution.

We propose the following scale, which starts at a level most would consider very coarsely correct and trends toward a very fine-grained notion of correct program execution.

1. not crashing the application
2. not raising any signals or externally noticeable events that were not visible before
3. externally visible behavior remains the same as previous
4. all behavior remains strictly the same as previous
5. correct with respect to the programmer's implementation (above)
6. correct with respect to the programmer's intent (may be poorly implementd)
7. correct with respect to the programmer's understanding (may be bad)
8. absolute correctness (where a programmer's mistake may be fixed)

Our evaluation will assess the performance of error virtualization and SEV against this scale.

2.4 Caveats and Limitations

Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed or even well-formed. Ideal computing systems would recover from attacks and errors without human intervention. However, the state of the art is far from mature, and most existing response mechanisms are external to the system they protect. Some simply crash the process that was attacked (and do nothing to fix the fault, thereby ensuring that the system is still vulnerable when it is rebooted). Other systems may restrict network connectivity or resource consumption.

The exploration of ways to bound the types of errors that arise in response to changing the semantics of program execution is an open area of research. This paper explores one method of shaping the behavior of the system towards an acceptable profile of failure dynamics.

Since our tools operate at the application level, they do not provide supervision or control for any operating system kernel code. We could potentially combine our tools with system call interposition or virtual-machine monitors to protect the kernel. The failure semantics of system calls are much more widely and better understood, and the return values for errors are more standard.

3 Implementation

Binary-level function behavior profiling is implemented in two systems: the open-source Valgrind (v 3.1.0) binary instrumentation framework, and the PIN dynamic binary translation software from Intel. Both frameworks allow tools or plug-ins to be written that inject code into a running application. The Valgrind tool is called Lugrind and is based on Weidendorfer's Callgrind[1] tool. We will describe the design and implementation of Lugrind; the PIN tool contains essentially the same functionality. Most of the implementation decisions were made to support SEV, but the core functionality can also be used for regression testing and host-based intrusion detection.

3.1 Basic Design

Binary-level function profiling is more difficult than may intially be expected. Lugrind makes minor changes to the Callgrind tool to take advantage of that tool's ability to track function entry and exit. We also incorporate a small change to Valgrind itself to expose the information that the tool requires, since the original instruction stream is not available to plug-ins – only the intermediate instruction representation is available (with good reason; tools should usually be portable. In the case of Lugrind, we

[1] http://kcachegrind.sourceforge.net/cgi-bin/show.cgi

require access to architectural-level information beyond the IR.).

The basic tasks that the profiler needs to accomplish are:

1. Record function entrances and exit events.
2. Record function return values.
3. Record call tree and function sequences.
4. Record argument values.
5. Calculate conditional probability between function sequences and return values
6. Schedule appropriate return values upon a fault or exploit

The system operates mainly in a mode that combines the training phase with the detection or self-healing phase. The system also provides the option to record and save a list of return values for each function learned from a previous run of the application under Lugrind. This latter feature can be used to bootstrap or preload the model in case the application had to be restarted.

The primary job of the tool is to maintain a collection of return values for each function. We discuss the use of the other information (argument values, call tree sequences, etc.) below. Assuming that the system is trained under an appropriate failure dynamics input set, the collection of return values for the function will reflect successful handling of strange, malformed, weird, outlandish, unlikely, and malicious input. If a fault manifests in a particular function, Lugrind then returns from that function with an return value selected from the collection. Future calls to that function can be "skipped" if Lugrind replaces the call to the function with an immediate return with a value selected from the return value collection for that function. Selection of return values can happen in several ways.

3.2 Return Value Selection

Selection strategies for an appropriate return value is referred to as "return value scheduling." Return value scheduling is an interesting problem and can be done in a few ways. First, we can simply select the first observed return value. Second, we can round-robin (FCFS) all observed return values (which has the potential for "euthanasia" of earlier, already-used return values). A variation of this scheme is delayed round-robin, where the current set is iterated through before newcomers to the set are considered. Third, we can rank-order return values by some priority. One priority metric is to match the input set with the closest input features seen so far. This ranking method (or really a lookup method) requires that the tool also maintain function argument values, or a suitable transformation thereof (hash, reversible, etc.). Another priority metric is to select a return value based on the current state of the call stack: if function c() always returns the value "5" when functions a() and b() proceed it on the call stack, and this configuration matches the call stack at the moment of healing, then 5 is a good candidate for SEV. Lugrind calculates the conditional probability of return values for an adjustable width window on the call stack.

An exploration of return value scheduling provides the motivation for continuous evaluation of the selected return value. That is, maintaining conditional probabilities matching certain call stack configurations with return values helps with prediction, but we should also provide a feedback mechanism to see how well subsequent configurations of the call stack match what we expect the call stack to look like given a particular return value. Performing this continuous evaluation helps the application by improving the quality of the prediction as well as providing a last line of defense for detecting a system that is still compromised despite the self-healing actions. Lugrind incorporates both pre-evaluation of call stack configurations (predict which return value is best based on sequence of calls) and post-evaluation (feedback on the decision).

After a return value has been scheduled and selected, an additional post-evaluation task is invoked. All return values for all functions are then placed in the tainted list and can be observed for differneces between them and the "untainted" mode. Depth of taint can be measured and is an indication of how many times the application has been self-healed.

3.3 Additional Control

In order to evaluate our system, we needed to incorporate additional control functionality into the tool. The main purpose of this functionality is to simulate the injection of errors into the application and the application's environment. This control plane can inject errors probabalistically or on demand. The effect of an error injection is to cause a particular function to fail and a return value to be selected to replace the function.

Besides probabalistic failure of a particular function, the control functionality can be given a message containing the name of a particular function to simulate failure for. This signal can specify failing the current function or a specific named function. In either case, the tool prints out which function it fails, some diagnostic information (e.g., how it selected a value), and the return value.

Since every function entry point is instrumented already, a small portion of that instrumentation is activated. This part of the instrumentation selects a return value, moves it to the return value register, and returns from the function.

This additional control functionality can be leveraged for more than system testing. The profiler can recieve a signal from some other host-based detector or monitor that the execution has been corrupted. It can then invoke the instrumentation for the appropriate function. Another option is to have only the control instrumentation running and dynamically start the heavier instrumentation upon notification from an external monitor. This can be done with Valgrind by invalidating the instruction cache. PIN is more elegant; it can attach to a running process.

```
...
  .> _setjmp(0x2E202020, ...)
   .> _setjmp(...) = 0
   .> main(0x2E202020, ...)
   . > aaa(0x2E202020, ...)
   . > aaa(...) = 42
   .> main(...) = 0
   .> exit(0x2E202020, ...)
   . > 0x009BDC71(0x2E202020, ...)
   . > 0x009BDC71(...) = 0
...
```

Figure 1: **Sample Lugrind Run Output Record.** *Lugrind maintains a record of all function calls and the call stack, including arguments and return values. As it records this information in its data structures, it outputs a log of the action. In this section of the log of a sample program run, the function `aaa()` returns the value 42.*

4 Related Work

Software self-healing is an active area of research and many systems draw on principles from intrusion detection, languages and compilers, and artificial immune systems. A major advantage of the SEV model and implementations is that we don't have to involve the programmer or the compiler. There is no application-level, library-level, or compiler-level functionality that needs to be built in or linked against. No OS support is needed. The programmer and compiler cannot anticipate all errors. Thus, source code or compiler transformations alone are unlikely to completely solve the problem, if at all, and cannot support legacy code or code which there is no source available. This restricts people from providing their own security especially for products like MS Windows, IExplorer, etc., in which the user must download a tool that monitors their computer without access to the application's source. We do not need to use more agressive source-code level techniques, which have severe limitations that render them less than useful for widespread adoption.

There are a number of approaches to providing self-healing software. Rinard *et al.* [9] have developed a compiler that inserts code to deal with access to unallocated memory by expanding the target buffer (in the case of writes) or manufacturing a value (in the case of reads). This technique is called *failure-oblivious computing* [9] [10].

The pH system [13] is an automatic reaction system based on the prinicplaes of an artificial immune system. The system is aimed at frustrating an attacker by using system call interposition to slow down an attacker's code.

The key idea of the Rx system [8] is to checkpoint the execution of a process and when an error occurs, rollback and replay the execution, but with the process's environment changed in a way that does not violate the API's its code expects. For example, the result of `malloc()` must be a buffer of at least the requested size, but that buffer may be located at a different offset than the original, be padded at both or either end, or be cleared to zero. This procedure is iterated over, with 'fixes' becoming more expensive, until execution proceeds past the detected error point. The error and the transparent environment fix are then recorded for the programmer to debug. This is a clever way to avoid the semantically incorrect fixes of failure oblivious computing and error virtualization.

Vigilante [1] is a system motivated by the need to contain rapid malcode (worms). To that end, Vigilante supplies a mechanism to detect an exploited vulnerability. It also defines a data structure for exchanging information about the discovered vulnerability. A major advantage of this vulnerability-specific approach is that Vigilante is exploit-agnostic and can be used to defend against polymorphic worms. While Vigilante doesn't address the self-healing of a piece of exploited software, it defines an architecture for production and verification of Self-Certifying Alerts (SCA's), a data structure for exchanging information about the discoverd vulnerability. Vigilante works by analyzing the control flow path taken by executing injected code.

DIRA [12] is a compiler extension that adds instrumentation to keep track of memory reads and writes and check the integrity of control flow transfer data structures. If the integrity fails, then the changed data is extracted and a network filter is created from it. Execution is recovered to a safe state.

Program shepherding [3] is a technique that validates control flow transfers at the machine level. As an optimization, it store the decision in a cache.

Song and Newsome [7] propose dynamic taint analysis for automatic detection of overwrite attacks. Tainted data is monitored throughout the program execution and modified buffers with tainted information will result in protection faults. Once an attack has been identified, signatures are generated using automatic semantic analysis. The technique is implemented as an extension to Valgrind and does not require any modifications to the program's source code but suffers from severe performance degradation.

FLIPS [4] is a system that attempts to self-heal by providing feedback to an anomaly detector to block confirmed code injection attacks as well as employing STEM's [11] error virtualization.

One of the most critical concerns with recovering from software faults and vulnerability exploits is ensuring the consistency and correctness of program data and state. An important contribution in this area is presented by Dempsky [2], which discusses mechanisms for detecting corrupted data structures and fixing them to match some pre-specified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate when faults were randomly injected.

5 Acknowledgements

We would like to thank the Valgrind developers, especially Nick Nethercote and Julian Seward for helpful advice with this and other Valgrind efforts. Josef Weidendorfer also deserves a special thanks for his contribution of the Callgrind tool (which jumps through a number of hoops to keep track of function entry and exit at the binary level) and his helpful advice on the construction of Lugrind.

6 Conclusions

Self-healing software is one autonomic defense technique for computing systems, but effective remediation strategies remain a challenge. Most approaches suffer from a lack of semantic correctness of the reaction mechanism. Error virtualization in particular needs to be improved.

We propose the Smart Error Virtualization (SEV) model and provide two systems that implement this model by profiling unmodified program binaries for function characteristics. A model of "failure dynamics" is learned for a variety of applications. A failure dynamics model captures the application's response to abnormal or malformed inputs and environment changes. It is this class of behavior that error virtualization should target and attempt to guide the application's state towards when an attack or fault has been detected.

Our tools can also be leveraged for host-based anomaly detection (similar to intrusion detection based on sequences of system calls) and regression testing.

References

[1] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)*, 2005.

[2] B. Demsky and M. C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[3] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[4] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 11^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 0–19, September 2005.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[6] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[7] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature

Generation of Exploits on Commodity Software. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.

[8] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)*, 2005.

[9] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In *Proceedings* 20^{th} *Annual Computer Security Applications Conference (ACSAC) 2004*, December 2004.

[10] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[11] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[12] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.

[13] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[14] D. Turner and S. Entwisle. Symantec Internet Security Threat Report. `http://enterprisesecurity.symantec.com/content.cfm?articleid=1539`, September 2004.