Return Value Predictability Profiles for Self–Healing

Michael E. Locasto[1], Angelos Stavrou[2], Gabriela F. Cretu[3], Angelos D. Keromytis[3], and Salvatore J. Stolfo[3]

[1] Institute for Security Technology Studies, Dartmouth College
[2] Department of Computer Science, George Mason University
[3] Department of Computer Science, Columbia University

Abstract. Current embryonic attempts at software self–healing produce mechanisms that are often oblivious to the semantics of the code they supervise. We believe that, in order to help inform runtime repair strategies, such systems require a more detailed analysis of dynamic application behavior. We describe how to profile an application by analyzing all function calls (including library and system) made by a process. We create predictability profiles of the return values of those function calls. Self–healing mechanisms that rely on a transactional approach to repair (that is, rolling back execution to a known safe point in control flow or slicing off the current function sequence) can benefit from these return value predictability profiles. Profiles built for the applications we tested can predict behavior with 97% accuracy given a context window of 15 functions. We also present a survey of the distribution of actual return values for real software as well as a novel way of visualizing both the macro and micro structure of the return value distributions. Our system helps demonstrate the feasibility of combining binary–level behavior profiling with self–healing repairs.

keywords: behavior profiling, anonamly detection, self-healing

1 Introduction

The growing sophistication of software attacks has created the need for increasingly finer-grained intrusion detection systems to drive the process of automated response and intrusion prevention. Because such fine-grained mechanisms are currently perceived as too expensive in terms of their performance impact, questions relating to the feasibility and value of such analysis remain unexplored. In particular, it seems that self–healing mechanisms would benefit from a detailed behavior profile.

This paper demonstrates the efficacy and feasibility of building profiles of application behavior at a fine-grained level of detail. We focus on the use of function return values as the main feature of these profiles. We do so because return values can help drive control flow decisions after a self–healing repair. In this paper, we show how to build profiles that contain this information at the binary level — that is, without making changes to the application's source, the OS, or the compiler.

1.1 Observing Program Behavior

A popular approach to observing program behavior utilizes anomaly detection on a profile derived from system call sequences [1–5]. Relatively little attention has been paid to the question of building profiles — in a non-invasive fashion — at a level of detail that includes the application's *internal* behavior. In contrast, typical system call profiling techniques characterize the application's interaction with the operating system. Because these approaches treat the application as a black box, they are generally susceptible[4] to mimicry attacks [7]. Furthermore, the increasing sophistication of software attacks [8] calls into question the ability to protect an application while remaining at this level of abstraction (*i.e.,* system call interface). Finally, most other previous approaches instrument the application's source code or perform static analysis. In contrast, we constructed a non–invasive return value collector tool using the Pin [9] dynamic binary rewriting framework to gather profile information.

1.2 Self–Healing

Various approaches to software self–healing [10–13] concentrate on a transactional approach in which the current sequence of functions is rolled back to some known safe point in execution [14, 11, 15], and the calculations done by the aborted "transactions" are undone. Such approaches require a profiling mechanism to both guide the selection of "known safe points" and set appropriate state at those points.

For example, the concepts of *error virtualization* [12] and *failure–oblivious computing* [10] are representative of approaches that attempt to execute through faults (*e.g.,* memory corruption due to an exploit) by manufacturing information that helps control subsequent execution. Failure–oblivious computing manufactures values for read operations and silently expands or truncates memory overwrites. In error virtualization, a heuristic helps determine the return value for an aborted function; the hope is that the rest of the software will gracefully handle this manufactured error return value and continue executing, albeit without the influence of the attacker.

Determining these return values employs source code analysis on the return type of the function in question. This approach is somewhat unsatisfactory; it seems as if it should be possible to dynamically and automatically collect enough information to determine appropriate error virtualization values. This paper addresses the problem of how to automatically extract enough information from program execution to accurately characterize program behavior in terms of return values to support self–healing.

Behavior profiling has been used to create policies for detection [16, 17]. In contrast, we suggest using this information to help automatically generate templates for repair policies [18]. In addition, this information can drive the selection of "rescue points" for the ASSURE system [15]. One goal of this paper is to provide systems like SEAD and ASSURE with a profiling mechanism.

[4] Gao *et al.* [6] discuss a measure of behavioral distance where sequences of system calls across heterogeneous hosts are correlated to help avoid mimicry attacks.

1.3 Caveats and Limitations

Binary-level function profiling proves more difficult than may initially be expected. Functions are source level artifacts that have only rough analogues at the machine level. Since a compiler can arbitrarily transform the source–level representation of a function or signal handling can interrupt control flow, it is difficult to cover all cases of function entry and exit. We rely on Pin [9] to detect these events, although it can fail to do so in the presence of tail recursion or aggressive function inlining by the compiler. Finally, because the profile is dependent on a particular binary, our system must recognize when an older profile is no longer applicable *e.g.,* as a result of a new version of the application being rolled out, or due a patch. We can detect this in several ways, including the modification time of the program image on disk.

1.4 Contributions

Overall, we demonstrate the utility of fine-grained application modeling to support self–healing repairs. Our work differs from related work (Section 2) on anomaly detection and self–healing software in two important respects: (1) the structure and granularity of our profiles, and (2) the focus on repair rather than detection.

We create a new model of program behavior extracted *dynamically* from the execution of the program binary without instrumenting the source code, modifying the compiler, or altering the OS. We condition this model based on a feature set that includes a mixture of parent functions and previous sibling functions. Prior approaches look at the call stack, thus ignoring previous siblings, which have already completed execution and so are no longer part of the call stack. This model can help select appropriate error virtualization values, inform the choice of rescue points, or drive the creation of repair policy templates. In addition, we provide a survey of return values used in real software. Finally, we propose *relative scaled k-means clusters*, a new way to simultaneously visualize both the micro and macro structure of feature-frequency behavior models. Details on our profiling experiments and results can be found in Section 4. Section 5 characterizes the return value content of the profiles.

2 Related Work

Our work provides a mechanism to describe application behavior. Thus, our modeling algorithm draws from a rich literature on host–based anomaly detection schemes. While this area is well–mined, we believe it is worthwhile to revisit previous efforts to validate and potentially improve on them. Most significantly, we focus on the utility of behavior profiles for post-attack repair rather than pre-attack detection.

Anomaly Detection Host-based anomaly detection is not a new topic. The seminal work of Hofmeyr, Somayaji, and Forrest [19, 3] helped initiate application behavior profiling at the system call level. Feng *et al.* [4] and Bhatkar *et al.* [20] contain good overviews of the literature in this space. Most approaches to host-based intrusion detection perform anomaly detection [2, 16, 5, 21] on sequences of system calls and their arguments [22]

because the system call interface represents the services that user–level malcode, once activated, must use to effect persistent state changes and other forms of I/O. System call information is easy to collect; the `strace(1)` and `ltrace(1)` tools for Linux are built to do exactly that. The closest work to our building of behavior profiles is the work by Mutz *et al.* [1] and Feng *et al.* [4]; the most significant differences in our model building is that we employ sibling functions when building profiles, and we examine the return values (rather than arguments). The most significant overall differences between our current work and the general space of system call AD is that we consider how to use this profile in the process of self–healing repairs.

Profiling for Self–Healing The key assumption underlying error virtualization [12] 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 existing 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.

ASSURE [15] attempts to minimize the likelihood of a semantically incorrect response to a fault or attack by identifying *error virtualization rescue points*: program locations that are known (or at least conjectured, according to a behavior profile) to successfully propagate errors and recover execution. The key insight is that a program should respond to malformed input differently than benign input; locations in the code that successfully handle these sorts of anticipated input "faults" are good candidates for recovering to a safe execution flow. We view our behavior profiling as a service provider to ASSURE's rescue point selection; ASSURE provides an input training set and handles the details of "teleporting" a failure to the appropriate rescue point.

3 Profile Structure

We define a profile structure that allows us to predict function return values based on the preceding context [23, 1] (functions that have just finished executing). Our system is a hybrid that captures aspects of both control flow (via the execution context) and portions of the data flow (via function return values). We construct an "execution context" for each function based on the application's behavior in terms of both control (predecessor function calls) and data (return values) flow. This context helps collapse *occurrences* of a function into an *instance* of a function to avoid under-fitting or over-fitting the model.

A behavior profile is a graph of execution history records. Each record contains an identifier, a return value, a set of arguments, and a context. Function names serve as identifiers (although callsite addresses are sometimes substituted). Parent and previous sibling functions compose the context. Argument and return values correspond to the values at function entrance and exit, respectively. The purpose of each item is to help identify an instance of a function. For example, considering every occurrence of `printf()` as the *same* instance reduces our ability to make predictions about its behavior. Likewise, considering all occurrences of `printf()` to be *distinct* instances reduces our ability to make predictions in a reasonable amount of time.

We adopt a mixture of parents and siblings to define a context for two reasons. First, a flat or nil context contains very little information to base a return value prediction on.

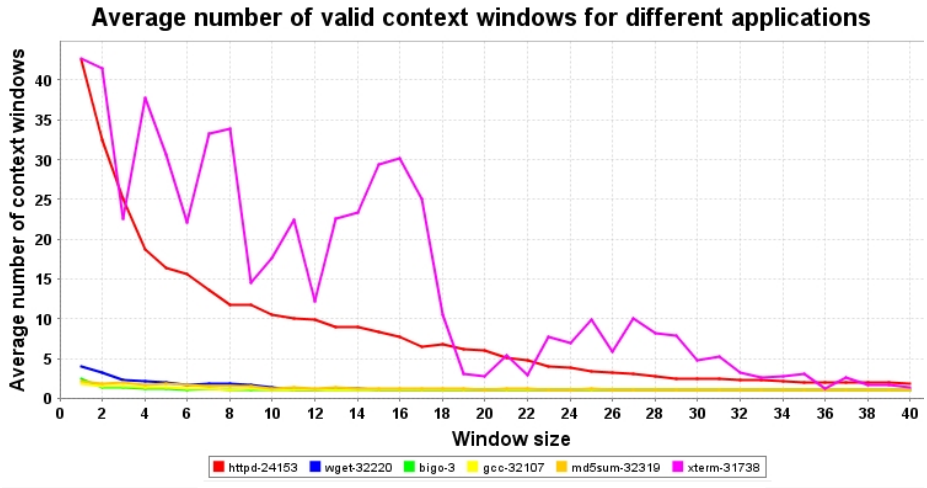


Fig. 1. *Drop in Average Valid Window Context.* This graph shows that the amount of unique execution contexts we need to store to detect changes in control flow decreases as window size increases. `xterm` is a special case because it executes a number of other applications. If we consider the ratio of valid context windows to all possible permutations of functions, then we would see an even sharper decrease.

Second, previous work focuses on the state of a call stack, which consists solely of parent functions. As our results in Section 4 demonstrate, the combination of parents and siblings is a powerful predictor of return values. The window size determines, for each function whose profile is being constructed, the number of functions preceding it in the execution trace that will be used in constructing that profile. Figure 1 provides insight: it shows that the amount of unique execution contexts drops as the window size increases. In contrast, if there were a large amount of valid windows, our detection ability would be diminished.

4 Evaluating Profile Generation

We start by assessing the feasibility of generating profiles that can predict the return values of functions. This section considers how to generate reliable profiles of application execution behavior for both server programs and command line utilities. These profiles are based on the binary call graph features combined with the return values of each function instance. We test and analyze applications that are representative of the software that runs on current server and desktop Unix environments, including: `xterm` (X.Org 6.7.0), `gcc` (GNU v3.4.4), `md5sum` (v5.2.1), `wget` (GNU v1.10.2), the `ssh` client (OpenSSH 3.9p1) and `httpd` (Apache/2.0.53). We also employ some crafted test applications to verify that both the data and the methods used to process them are correct. We include only one of these applications (`bigo`) here because it is relatively small, simple to understand, and can easily be compared against profiles obtained from

the other applications. The number of unique functions for all these applications is: xterm, 2111; gcc, 294; md5sum, 239; wget, 846; ssh, 1362; httpd, 1123; and bigo, 129.

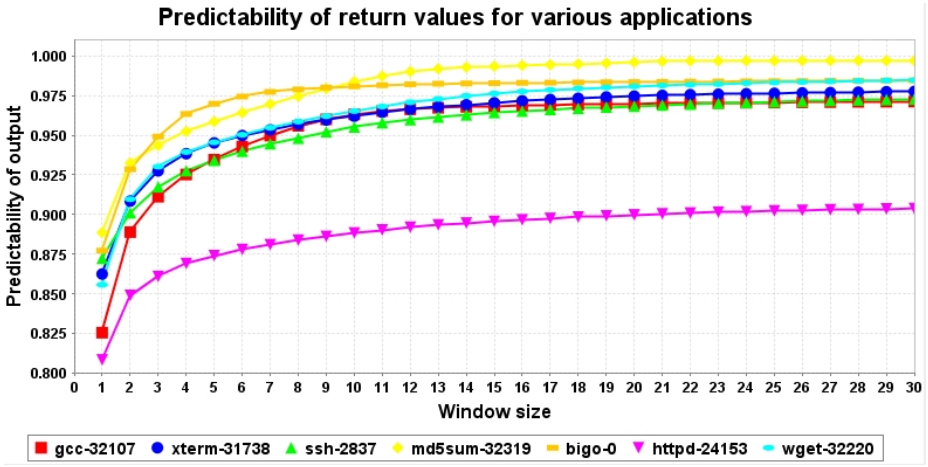
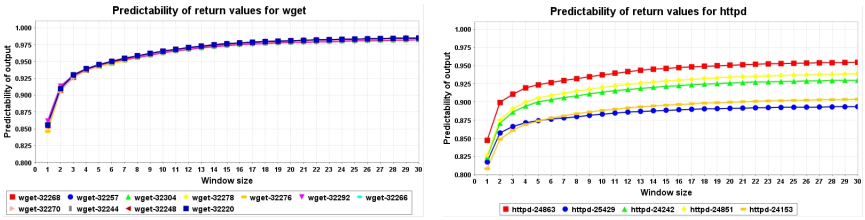


Fig. 2. *Average Predictability of Return Values*. Return value prediction for various applications against a varying context window size. Window sizes of 15 or more achieve an average prediction of 97% or more for all applications other than `httpd` (with a rate of about 90%).

Return Value Prediction Finding a suitable repair for a function, in the context of the self–healing systems we have been discussing, entails examining the range of return values that the function produces. As Section 3 explains, the notion of return value "predictability" is defined as a value from 0..1 for a specific context window size. A predictability value of 1 indicates that we can fully predict the function's return value for a given context window.

Figure 2 shows the average predictability for the set of examined applications. It presents a snapshot of our ability to predict the return value for various applications when we vary the context window size (*i.e.,* the history of execution). Using window sizes of more than 15 can achieve an average prediction rate of more than 97% for all applications other than `httpd`. For `httpd`, prediction rates are around 90%. This rate is mainly caused by Apache's use of a large number of custom functions that duplicate the behavior of standard library functions. Moreover, Apache is larger and more complex than the other applications and has the potential for more divergent behavior. Of course, this first data set is only a bird's eye view of an overall trend since it is based on the behavior of the average of our ability to predict function return values.

To better understand how our predictions perform, we need to more closely examine the measurements for different runs of the same application. In Figure 3(b) and Figure 3(a) we present results for different runs of `httpd` and `wget`. The `wget` utility was executed using different command line arguments and target sites. Apache was used as a daemon with the default configuration file but exposed to a different set of requests



(a) *Average Predictability of Return Values for Different Runs of* `wget`. Although there are 11 runs for `wget`, each individual run is both highly predictable (>98%) and very similar to the others' behavior for different window sizes.

(b) *Average Predictability of Return Values for* `httpd` *and for Different Runs*. Although return value prediction remains high (>90%) for `httpd`, some variations are observable between the different runs. This phenomena is encouraging because it suggests that the profile can be specialized to an application's use at a particular site.

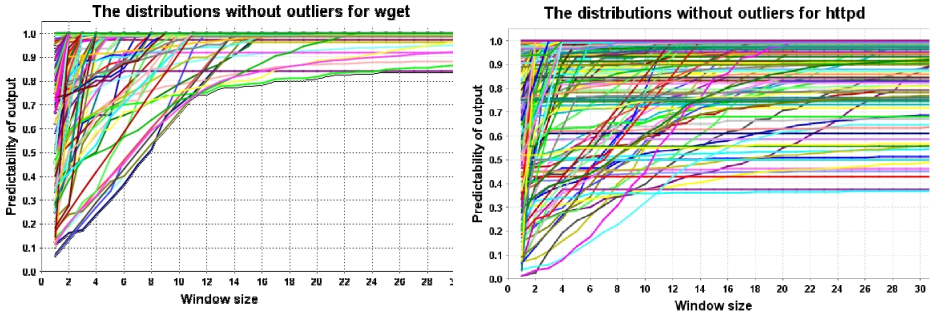
Fig. 3. Return Value Predictability for both `wget` and `httpd` with Different Window Sizes

for each of the runs. As we expected, `wget` has similar behavior between different runs: both the function call graph and the generated return values are almost identical. On the other hand, Apache has runs that appear to have small but noticeable differences. As reflected in the average plots, however, all runs still have high predictability.

Some questions remain, including how effective our method is at predicting return values of individual functions. Also, if there are any function that we cannot predict well, how many functions of this type are there, and is there some common feature of these functions that defies prediction? Answering these questions requires measurements for individual function predictability. To visually clarify these measurements, in Figures 4(a) and 4(b), we remove functions that have a prediction of two or more standard deviations from the average. The evolution of predictability for `wget` and `httpd` is illustrated in Figure 4(a) and Figure 4(b). This evolution is consistent with what we observe in Figure 2: most functions are predictable — and for small context windows.

A small percentage of the functions, however, produce return values which are highly unpredictable. This situation is completely natural: we cannot expect to predict return values that depend on runtime information such as memory addresses. Additionally, there are some functions that we *expect* to return a non-predictable value: a random number generator is a simple example. In practice, as we can deduce from our experiments (see Table 1), the number of such "outlier" functions is rather small in comparison to the total number of well–behaved, predictable functions.

In Table 1, each column represents the percentage (out of all functions in each program) of common or outlier functions. The first column presents the percentage of functions that are outliers *within* a program: that is, functions that deviate from the average profile by more than two standard deviations for all windows of size >10. For each



(a) *Predictability of Return Values for* `wget` *Functions*. Each line represents a function and its predictability evolution as context window size increases. Most functions stabilize after a window size of ten. This graph excludes a small set (Table 1) of outlier functions (functions that are two standard deviations from the average).

(b) *Predictability of Return Values for* `httpd` *Functions*. Each line represents a function and its predictability evolution as context window size increases. As expected, `httpd` has more functions that diverge from the average. It also has more outliers, as shown by Table 1.

Fig. 4. Per-Function Return Value Predictability for both `wget` and `httpd`

program, there are relatively few "outlier" functions. The second column examines the percentage of common outliers: outliers that appear in two or more applications. We can see that the functions that are unpredictable are consistent and can be accounted for when creating profiles. The third column displays non-outlier functions that are common across applications. These common and predictable functions help show that some aspects of program behavior are consistent across programs.

5 Return Value Characteristics

While Section 4 shows how well we can predict return values, this section focuses on *what* return values actually form part of the execution profile of real software, and how those values are embedded throughout the model structure. We wrote a Pin tool to capture the frequency distribution of return values occurring in a selection of real software, and we created distributions of these values. These distributions provide insight into both the micro and macro structure of a return value behavior profile. Our data visualization technique scales the height and width of each cluster to simultaneously display both the intra- and inter-cluster structure. Our analysis aims to show that return values can reliably classify similar runs of a program as the same program as well as distinguish between execution models of different programs.

Return Value Frequency Models Our Pin tool intercepts the execution of the monitored process to record each function's return value. The tool builds a table of return value

Table 1. *Percentage of Unpredictable (Outlier) Functions.* We illustrate the nature of the overlap in behavior profiles by examining which functions are outliers both within and across programs.

Application	% outliers	% common outliers	% common functions
gcc	5	53	51
md5sum	5	87	84
wget	6	62	56
xterm	6	39	17
ssh	5	35	33
httpd	10	10	28

frequencies. After the run of the program completes, we feed this data to MATLAB for a further evaluation that leads to a final return value frequency model. As a proof of concept, a model for a particular monitored process is simple, consisting of the average frequency distribution over multiple runs of the same program. Intuitively, we expect several types of clusters to emerge out of the average frequency distribution. We anticipate clusters that contain very high frequency return values, such as -1, 0, and 1 (standard error or success values as well as standard output handles). We expect a larger, more dispersed cluster that records pointer values as well as a cluster containing more "data" values such as ASCII data processed by character or string handling routines.

Table 2. *Manhattan Distance Within and Between Models.* The diagonal (shown in italics) displays the average distance between each trace and the behavior profile derived from each trace of that program. All other entries display the distance between the execution models for each program. We omit the lower entries because the table is symmetric. Note the difference between gzip and gunzip as well as the similarity of gzip to itself.

	date	echo	gzip	gunzip	md5sum	sha1sum	sort
date	*3.03e+03*	3.72e+03	1.61e+07	1.87e+06	6.46e+04	6.47e+04	5.45e+03
echo	-	*548*	1.61e+07	1.87e+06	6.41e+04	6.42e+04	5.43e+03
gzip	-	-	*212.4*	1.79e+07	1.61e+07	1.61e+07	1.61e+07
gunzip	-	-	-	*1.91e+04*	1.92e+06	1.92e+06	1.87e+06
md5sum	-	-	-	-	*3.03e+04*	3.38e+04	6.56e+04
sha1sum	-	-	-	-	-	*1.67e+04*	6.57e+04
sort	-	-	-	-	-	-	*4.24e+03*

We examine three hypothesis dealing with the efficacy of execution behavior profiles based on return value frequency:

1. traces of the same program under the same input conditions will be correlated with their model
2. the model of all traces of one program can be distinguished from the model of all traces of another program
3. we can make the structure of the return value frequency models apparent using k-means clustering

For the first hypothesis, we use Manhattan distance as a similarity metric in order to compare each trace of the same process with the return value model of that process. In effect, we compare each return value frequency to the corresponding average frequency among all traces of that program. To evaluate the second hypothesis, we use the Manhattan distance between each process model. The base set for the return values consists of all return values exhibited by all processes that are analyzed over all their runs. Table 2 shows how each model for a variety of program types (we include a variety of programs, like sorting, hashing, simple I/O, and compression) stays consistent with itself under the same input conditions (smaller Manhattan distance) and different from models for each other program (larger Manhattan distance).

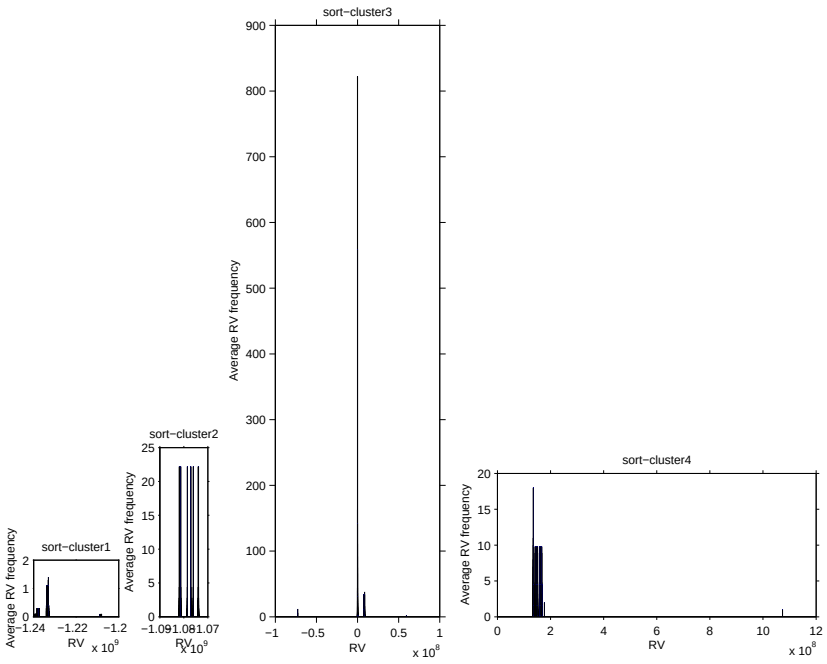


Fig. 5. *Relatively Scaled k-means Clusters for* `sort`. Note how each component of the model is scaled relative to the others while displaying the distribution of similar-frequency values internally; this technique clearly displays the differences between the high frequency return values (cluster 3) and the frequent but more widespread parts of the model (cluster 4) as well as the behavior of values within each component. Both the vertical and horizontal dimensions of each cluster are scaled. We display the clusters in increasing order from left to right determined by the lower end of the horizontal axis range.

Each process has a particular variance with each trace quantified in the similarity value between its model and the trace itself, but when compared against the rest of the processes it can be easily distinguished. We ran each program ten times under the same input profile to collect the traces and generate the model for each program. We used as input profiles generic files/strings that can be easily replicated (in some cases no input was needed): date - N/A, echo - "Hello World!", gzip - *httpd-2.2.8.tar*, gunzip - *httpd-2.2.8.tar.gz*, md5sum *httpd-2.2.8.tar*, sha1sum - *httpd-2.2.8.tar* and sort - *httpd.conf* (the unmodified config file for httpd-2.2.8).

Clustering with k-means Section 4 shows how to build profiles that are useful in predicting return values. The analysis here aims to achieve a better idea of what those actual return values are and how frequently real applications use them. We cluster the return values into frequency classes, and we chose the k-means method to accomplish the clustering. The large disparity in the magnitude of return value frequencies can reduce our ability to convey information about the overall structure of the model if displayed in a simple histogram. Accordingly, we found a new way to simultaneously display both the internal structure of each cluster as well as the external relationships between the clusters. Our clustering method captures the localized view of return value frequency per RV region and our visualization method provides insight into the relative coverage of a particular RV region. Figures 5, 6(a), 6(b), 7(a), 7(b), 8(a), and 8(b) present the clusters obtained for each of the analyzed processes. Each model has a predominant cluster (*e.g.,* cluster2 for data, cluster2 for gzip, *etc.*) which contains the discriminative return value frequencies and has coverage. The remaining clusters contain the lower frequency return values. We conjecture that by increasing the number of cluster we can achieve better granularity that can distinguish between different types of return values and classify them accordingly.

6 Conclusion

We propose a novel approach to dynamically profiling application execution behavior: modeling the return values of internal functions. Our return value sniffer is available under the GNU GPL at our website[5]. We show that using a window of return values, including values returned by sibling and parent functions, can make return value prediction as accurate as 97%. We also introduce a novel visualization method for conveying both the micro and macro structure of the return value frequency model components. We intend to investigate models that operate independently of the input profile. We intend to investigate how our behavior profiling mechanism can be used to create repair policy and assist other self–healing systems select an appropriate response.

Acknowledgments

This paper reports on work that was supported in part by ARO/DHS contract DA W911NF-04-1-0442, USAF/AFRL contract FA9550-07-1-0527, and NSF Grant 06-

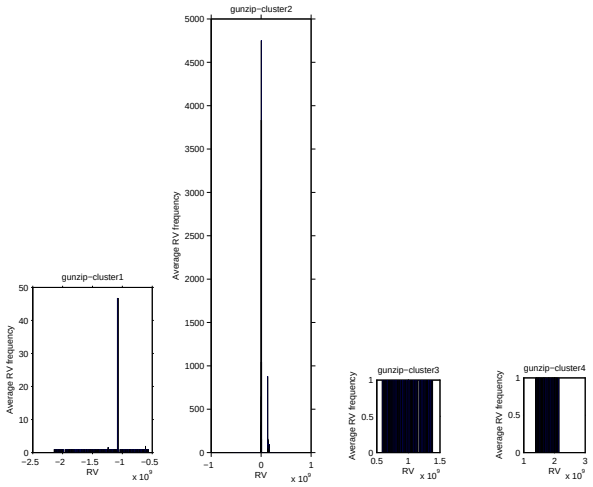
[5] `http://www.cs.dartmouth.edu/~locasto/research/rval/`

27473. The content of this work is the responsibility of the authors and should not be taken to represent the views or practices of the U.S. Government or its agencies.

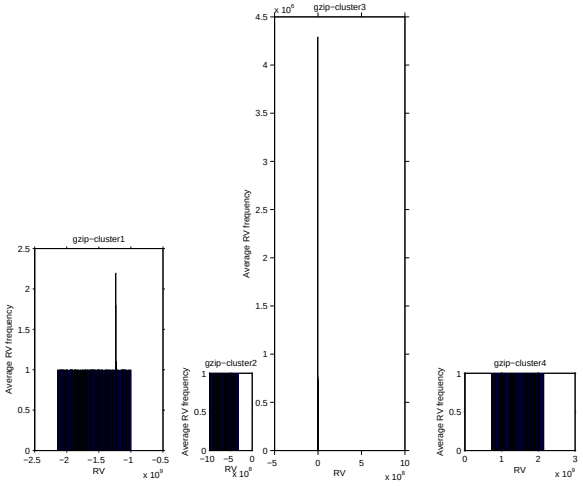
References

1. Mutz, D., Robertson, W., Vigna, G., Kemmerer, R.: Exploiting Execution Context for the Detection of Anomalous System Calls. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID). (2007)
2. Chari, S.N., Cheng, P.C.: BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In: Proceedings of the 9^{th} Symposium on Network and Distributed Systems Security (NDSS 2002). (2002)
3. Somayaji, A., Forrest, S.: Automated Response Using System-Call Delays. In: Proceedings of the 9^{th} USENIX Security Symposium. (August 2000)
4. Feng, H.H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy. (May 2003)
5. Gao, D., Reiter, M.K., Song, D.: Gray-Box Extraction of Execution Graphs for Anomaly Detection. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2004)
6. Gao, D., Reiter, M.K., Song, D.: Behavioral Distance for Intrusion Detection. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005) 63–81
7. Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (November 2002)
8. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-Control-Data Attacks Are Realistic Threats. In: Proceedings of the 14^{th} USENIX Security Symposium. (August 2005) 177–191
9. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood., K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of Programming Language Design and Implementation (PLDI). (June 2005)
10. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., W Beebee, J.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: Proceedings 6^{th} Symposium on Operating Systems Design and Implementation (OSDI). (December 2004)
11. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: 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)
12. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a Reactive Immune System for Software Services. In: Proceedings of the USENIX Annual Technical Conference. (April 2005) 149–161
13. Smirnov, A., Chiueh, T.: DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In: Proceedings of the 12^{th} Symposium on Network and Distributed System Security (NDSS). (February 2005)
14. Brown, A., Patterson, D.A.: Rewind, Repair, Replay: Three R's to dependability. In: 10^{th} ACM SIGOPS European Workshop, Saint-Emilion, France (September 2002)
15. Sidiroglou, S., Laadan, O., Keromytis, A.D., Nieh, J.: Using Rescue Points to Navigate Software Recovery (Short Paper). In: Proceedings of the IEEE Symposium on Security and Privacy. (May 2007)

16. Provos, N.: Improving Host Security with System Call Policies. In: Proceedings of the 12^{th} USENIX Security Symposium. (August 2003) 207–225
17. Lam, L.C., cker Chiueh, T.: Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In: Proceedings of the 7^{th} International Symposium on Recent Advances in Intrusion Detection. (September 2004)
18. Locasto, M.E., Stavrou, A., Cretu, G.F., Keromytis, A.D.: From STEM to SEAD: Speculative Execution for Automatic Defense. In: Proceedings of the USENIX Annual Technical Conference. (June 2007) 219–232
19. Hofmeyr, S.A., Somayaji, A., Forrest, S.: Intrusion Detection System Using Sequences of System Calls. Journal of Computer Security **6**(3) (1998) 151–180
20. Bhatkar, S., Chaturvedi, A., Sekar., R.: Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In: Proceedings of the IEEE Symposium on Security and Privacy. (2006)
21. Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-Sensitive Intrusion Detection. In: Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005)
22. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous System Call Detection. ACM Transactions on Information and System Security **9**(1) (February 2006) 61–93
23. Eskin, E., Lee, W., Stolfo, S.J.: Modeling System Calls for Intrusion Detection with Dynamic Window Sizes. In: Proceedings of DARPA Information Survivabilty Conference and Exposition II (DISCEX II). (June 2001)

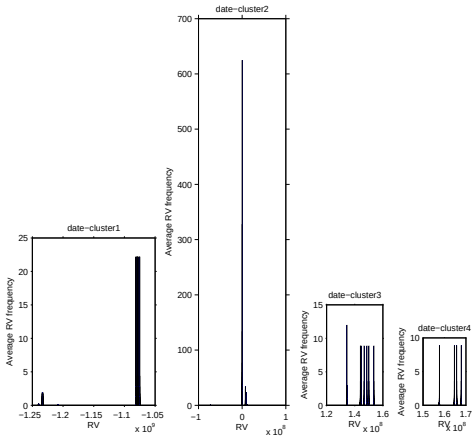


(a) *k-means cluster for* `gunzip` *return values.*

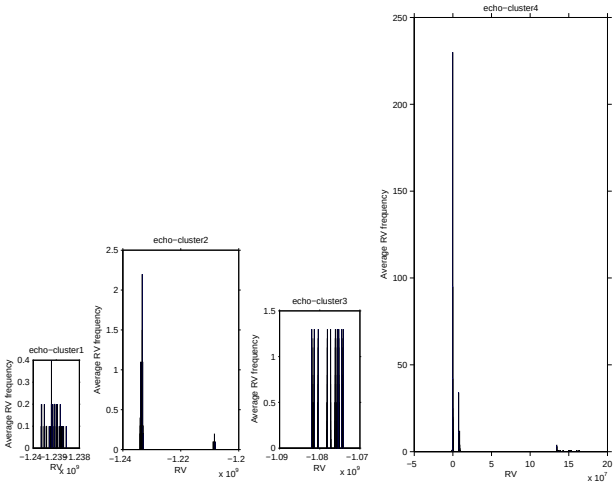


(b) *k-means cluster for* `gzip` *return values.*

Fig. 6. Return Value Frequency Distributions for a Compression Program

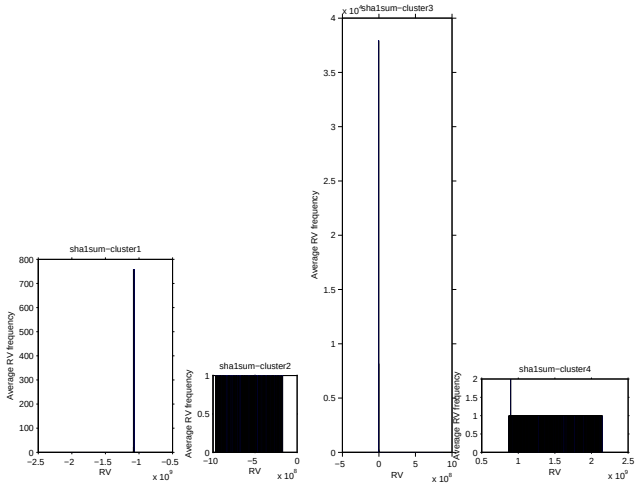


(a) *k-means cluster for* `date` *return values.*

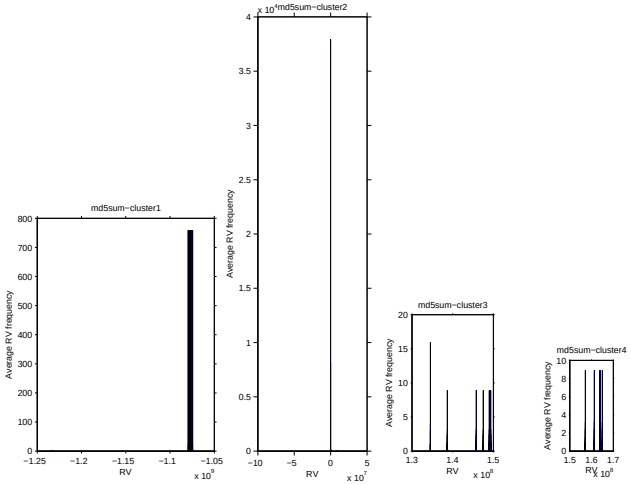


(b) *k-means cluster for* `echo` *return values.*

Fig. 7. Return Value Frequency Distributions for Output Programs



(a) *k-means cluster for* `sha1sum` *return values*.



(b) *k-means cluster for* `md5sum` *return values*.

Fig. 8. Return Value Frequency Distributions for Hash Programs