Quantifying Application Behavior Space for Detection and Self-Healing

Michael E. Locasto   Angelos Stavrou   Gabriela F. Cretu   Angelos D. Keromytis
Salvatore J. Stolfo

Abstract

The increasing sophistication of software attacks has created the need for increasingly finer-grained intrusion and anomaly detection systems, both at the network and the host level. We believe that the next generation of defense mechanisms will require a much more detailed dynamic analysis of application behavior than is currently done. We also note that the same type of behavior analysis is needed by the current embryonic attempts at self-healing systems. Because such 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 and unanswered.

We present a new mechanism for profiling the behavior space of an application by analyzing all function calls made by the process, including regular functions and library calls, as well as system calls. We derive behavior from aspects of both control and data flow. We show how to build and check profiles that contain this information at the binary level – that is, without making changes to the application's source, the operating system, or the compiler. This capability makes our system, *Lugrind,* applicable to a variety of software, including COTS applications. Profiles built for the applications we tested can predict behavior with 97% accuracy given a context window of 15 functions. Lugrind demonstrates the feasibility of combining binary-level behavior profiling with detection and automated repair.

1 Introduction

A key problem in information security is the inability of systems to automatically protect themselves from attack. In order to have a reasonable chance at surviving or deflecting current and emerging attacks, a system must incorporate automatic reaction mechanisms. There is an growing interest in self-healing software as a solution to this problem [25, 23, 26, 21, 6, 17]. However, these systems still require a sensor to detect when the self-healing mechanisms should be invoked. A popular approach to host-based sensors is anomaly detection based on a profile derived from sequences of system calls.

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-based profiling techniques characterize the application's interaction with the operating system. Because these approaches treat the whole application as a black box, they are generally susceptible to mimicry attacks [29]. Furthermore, the increasing sophistication of software attacks [5] 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 advocate building a profile from a very fine-grained level of detail obtained in a non-invasive fashion, operating on unmodified application binaries at runtime. The primary focus of this paper is on demonstrating the efficacy and feasibility of building this type of profile. However, we also consider the design, construction, and preliminary performance evaluation of Lugrind, a hybrid detection and repair system. Lugrind is responsible for collecting the profile data, detecting deviations from that profile, correcting (or "self-healing") those deviations, and validating that subsequent behavior matches the learned profile.

1.1 Motivation: Self-Healing and (Smart) Error Virtualization

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* [25] and *failure-oblivious computing* [23]. While both of these approaches showed promising results as far as supporting system survivability and preventing the exploitation of vulnerabilities by transparently masking failures (which can be as diverse as illegal memory dereferences, divisions by zero, and buffer overflows), they suffer from the potential for semantically incorrect execution. Such a shortcoming is devastating for applications that perform precise (*e.g.,* scientific, financial, *etc.*) calculations or provide authentication or authorization services. In addition, the heuristics for error virtualization fail about 12% of the time for the tested applications. Even for applications that require continued availability, this failure rate is quite discouraging.

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. These return values were determined by employing simple heuristics based on the return type of the offending function as determined by source code analysis.

The programs tested in Sidiroglou *et al.* [25] 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 proposed system (STEM) requires access to the source code of the application. These shortcomings motivate the development of a tool that can operate on an unmodified application binary and learn appropriate "error virtualization" values during runtime. We term the process of learning these values Smart Error Virtualization (SEV). These return values are selected using a conditional probability based on matching of the execution history at repair time with similar configurations encountered during training. This paper describes the design, implementation, and evaluation of such a system.

1.2 Goals and Contributions

Our primary goal is to provide a way to specify the phase space of application behavior for both detection and self-healing. The most notable contribution of Lugrind is that it integrates all three critical stages of intrusion defense: detection, repair, and repair validation. Lugrind can be differentiated from previous work on host-based anomaly detection and self-healing software by the following three contributions:

1. a unified model of program behavior extracted *dynamically* from the execution of the program binary without instrumenting the source code, modifying the compiler, or altering the operating system. Previous work usually examines system calls or is driven by static analysis. We constructed a Valgrind–based [16] tool, Lugrind, for extracting this model, which includes application and library function calls as well as system calls.

2. conditioning of 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.

3. the notion of Smart Error Virtualization (SEV), a technique for self-healing that involves learning appropriate function return values at runtime.

One of the most important implications of this unified model is that the information it contains can be concurrently leveraged for three different purposes: detection of program misbehavior (*i.e.,* deviation from a learned profile), automatic repair of said misbehavior, and automatic validation of those repairs. We discuss the design and implementation of Lugrind in Section 3 and Section 4, respectively. We leverage these contributions to perform two empirical studies. The first study builds profiles for a variety of applications and examines the correctness of the error virtualization hypothesis. The second study is a preliminary performance analysis of the system aimed at showing the feasibility of the approach on real software. The runtime (detection) overhead relative to binary translation ranges from as little as a 6% decrease in performance to a 3X slowdown, with most applications experiencing a 22% to 35% penalty. Although this cost seems prohibitive for most production environments, we believe that it represents a promising starting point. We intend to investigate several techniques that can be used to reduce the performance penalty of our approach. However, our goal in this paper is demonstrate the *usefulness* of fine-grained application modeling, for both detection and self-healing purposes. More details on our hypothesis, experiments, and results can be found in Section 5.

2 Related Work

Our work aims to provide a unified mechanism to describe application behavior. This description can be employed for three purposes: (a) as a profile for detection of application misbehavior, (b) as a model to aid automatic response by self-healing, and (c) as a model to guide automated testing to ensure that the self-healing does not cause the application to deviate further from its known behavior. The mechanism thus draws from a rich literature on host-based anomaly detection schemes and self-healing software.

2.1 Host-Based Intrusion Detection

Host-based anomaly detection is not a new topic. Starting with the seminal work of Hofmeyr, Somayaji, and Forrest [12, 27], profiling an application has previously been explored by examining the application's behavior at the system-call level [4]. Such system-call level information is easy to collect; the `strace` tool for various flavors of Unix is built to do exactly this task. While previous work was highly novel, one may be tempted to ask why we need Lugrind – potentially yet another call-stack oriented, host-based anomaly detection sensor, especially when previous work includes systems as effective and well-structured as that described by Gao *et al.* [9].

Our research is motivated by two observations. First, it is worthwhile to revisit previous efforts to validate and potentially improve on them. Second, our system includes a number of differences from previous work that collectively represents a major shift in the construction of host-based intrusion defense systems.

Most previous approaches to host-based ID perform anomaly detection on sequences of system calls with slight variations in flavor (for example, considering the settings of environment or configuration variables [11]). The work of Feng *et al.* [8] includes an excellent overview of the literature circa 2003. The work of Bhatkar *et al.* [1] also contains a good overview of the more recent literature and makes the point that previous approaches concentrate mostly on the integrity of control flow. To complement this work, they provide a technique for *dataflow* anomaly detection. 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). Although we focus on return values, our system is also capable of capturing function argument values, but we defer analysis of this type of data for future work.

Most host-based anomaly detectors are susceptible to mimicry attacks [29]. Gao, Reiter, and Song [10] discuss a measure of behavioral distance for intrusion detection where sequences of system calls across different (possibly heterogeneous) hosts are correlated to help avoid mimicry attacks. Although we have not tested this hypothesis, we believe that the profiles Lugrind generates are resistant to mimicry attacks because they incorporate both data and control flow information at a finer granularity than previous work.

The pH system [27] foreshadows the development of automatic reaction systems. Its aim is to frustrate an attacker by using system call interposition to slow down an attacker's code. While not strictly self-healing, this system was among the first to propose an active reaction mechanism to foil attacks and is representative of the seminal work in artificial immune systems.

2.2 Automatic Response

Software self-healing is an active area of research. Rinard *et al.* [22] have developed compiler extensions that 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). DIRA [26] 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. There are a number of systems that are closely related to DIRA's basic goals. Liang and Sekar [14] and Xu *et al.* [31] concurrently propose using address space randomization to drive the detection of memory corruption vulnerabilities and create a signature to block further exploits of this type.

The key idea of the Rx system [21] is to checkpoint the execution of a process in anticipation of system errors. When an error is encountered, execution is rolled back and replayed, but with the process's environment changed in a way that does not violate the API's its code expects. This procedure is repeated with different environment alterations until execution proceeds past the detected error point. This procedure is a clever way to avoid the semantically incorrect fixes of failure oblivious computing and error virtualization.

ASSURE [24] is a novel attempt to minimize the likelihood of a semantically incorrect response to a fault or attack. The authors propose the notion of *error virtualization rescue points*. A rescue point is a program location that is known to successfully propagate errors and recover execution. The key insight is that a program will respond to malformed input differently than legal input; locations in the code that successfully handle these sorts of anticipated input "faults" are good candidates for recovering to a safe execution flow. This technique is combined with operating system virtualization [19] to help ensure that recovery actions do not interfere with the normal operation of the system. As part of their strategy for identifying rescue points and determining the corresponding error virtualization value, the authors employ a tool based on Dyninst [2] that is similar in nature to Lugrind's data collection capabilities. In terms of self-healing systems, we view the two approaches as complementary: we seek the best return value to use *at the function that fails or behaves in an abnormal way;* ASSURE seeks *the best location* to which to "teleport" a failure that is detected elsewhere in the program. Note that the fine-grained models of application behavior that we build with Lugrind are used not just for self-healing but also for anomaly detection.

Vigilante [6] is a system motivated by the need to contain rapid malcode. Vigilante supplies a mechanism to detect an exploited vulnerability (by analyzing the control flow path taken by executing injected code) and defines a data structure (Self-Certifying Alert) for exchanging information about this discovery. A major advantage of this vulnerability-specific approach is that Vigilante is exploit-agnostic and can be used to defend against polymorphic worms.

Song and Newsome [18] propose dynamic taint analysis to monitor for injection attacks. The technique is implemented as an extension to Valgrind and does not require any modifications to the program's source code. The authors extend the system [17] with vulnerability-specific execution filters (VSEF), an idea with a different mechanism, but similar goals to Shield [30].

3 Approach

Lugrind is a binary-level application profiler that has three main use cases:

- detection of deviations from a learned profile for **host-based anomaly detection** or **regression testing**
- **software self-healing** by employing Smart Error Virtualization (SEV)
- **automatic repair validation** to ensure that self-healing effectively addresses the attack

A major advantage of Lugrind is that the programmer or the compiler does not need to be involved; neither entity can anticipate all errors. Source code transformations alone are unlikely to completely solve the problem and are not applicable when source is not available. Finally, the use of source-based tools prevents end users from providing their own security, especially for COTS applications.

The logical goal of application behavior profiling is to create policies for detection and self-healing. Automatic extraction of system call policies is covered by Provos [20] and Lam and Chiueh [13]. This section describes the overall design strategy for Lugrind, how we define and construct a profile, and the features this profile incorporates. The central idea is to construct a profile that allows us to predict function return values based on the preceding context.

3.1 Profile Definition and Building

An application's profile is a list of cross products of three items: a Function Identifier (*e.g.,* $\{f_0,\ f_1,\ f_2\ \ldots\ f_n\}$), a Profile Feature (*e.g.,* {PARENT, SIBLING, MIX, FLAT}), and the window size for the Profile Feature. The FI is a unique ID (typically, this ID is the function name, but addresses/call sites can also be used) extracted from the application at runtime. The PF is one of four features indicating the mixture of execution context. Execution context can be nil (FLAT), consist of only parents (PARENT), consist of just previous siblings (SIBLING), or a mixture of parents and previous siblings (MIX). We adopt the last feature in this paper for a number of reasons. First, a FLAT context contains very little information to base a return value prediction on. Second, previous work focuses on the state of a call stack, which consists solely of a PARENT context. Therefore, we choose to mix previous SIBLING functions into the PARENT feature. As our results in Section 5 demonstrate, this combination is a powerful predictor of return values. The window size determines, for each FI 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 shows an example window for a particular function in that execution trace.

The purpose of each item in a profile is to uniquely identify an instance of a function. The profile's feature set helps "unflatten" the function namespace of an application; for example, `printf()` appears many times with many different context windows and return values, making it hard to characterize. Considering every occurrence of `printf()` to be the same instance reduces our ability to make predictions about its behavior. On the other hand, considering all occurrences of `printf()` to be separate instances combinatorially increases the space of possible behaviors and similarly reduces our ability to make predictions about

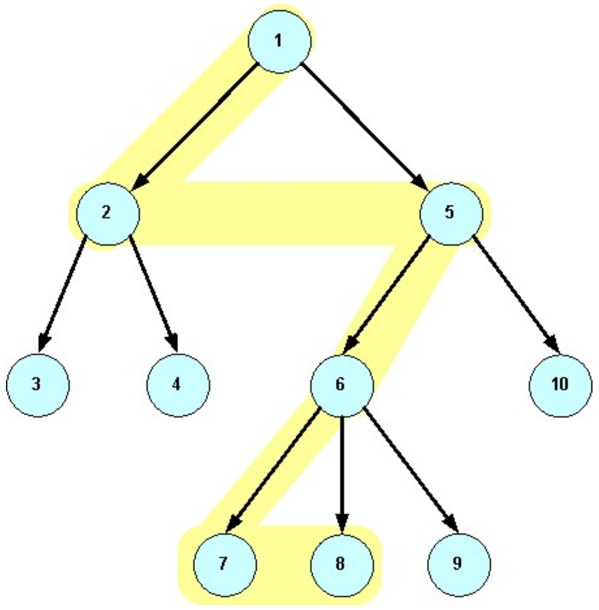

Figure 1: *Example of Computing Execution Window Context.* Starting from function 8, we traverse the graph beginning from the previously executed siblings up to the parent. We recursively repeat this algorithm for the parent until we either reach the window width or the root. In this example, the window contains functions 7, 6, 5, 2, 1. Systems that examine the call stack would only consider 6, 5, and 1 at this point.

its behavior in a reasonable amount of time. Therefore, we need to 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.

3.2 Training for "Predictability" of Return Values

Profile creation is implemented by using the MIX feature, as illustrated in Figure 1. During training, the system learns which return values to predict based on execution contexts of varying window sizes. The general procedure attempts to compute the prediction score by iteratively increasing the window size and seeing if additional information is revealed by considering the extra context.

We define the return value "predictability score" as a value from zero to one. For each window leading to a function, we calculate what we call the individual score: the relative frequency of this particular window when compared with the rest of the windows leading to that function. The predictability score for a function F is the sum of the individual scores that lead to a single return value. An example of this procedure is shown in Figure 2. We do not consider windows that contain smaller windows that lead to a single return value since the information that they impart is already subsumed by the smaller execution context. For example, in Figure 2 we do not consider all windows with a suffix of AF (*i.e.,* $*AF$). Since the predictability score is a non-decreasing function of the window size, increasing the context window size can only gain information and add to the overall predictability score.

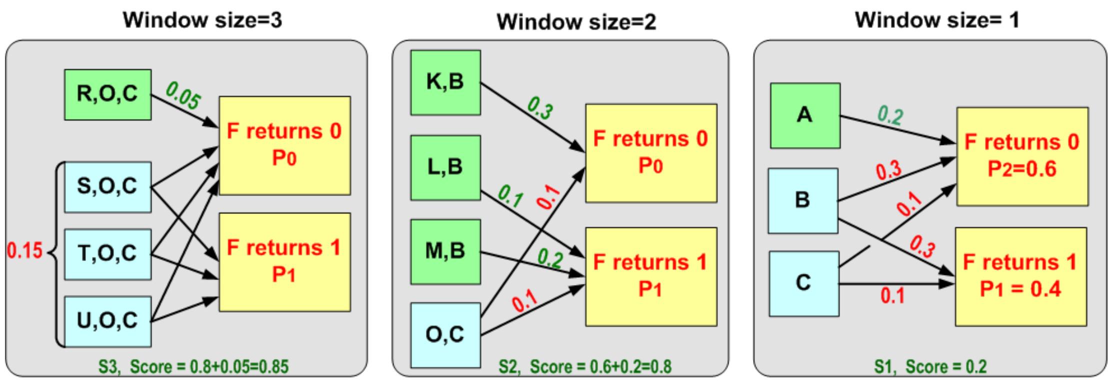

Figure 2: *Example of Computing Return Value Predictability (predictability score).* The figure illustrates the procedure for function F and for two return values 0 & 1 for three window sizes. The arrow labels indicate what percentage of instances for the given window will lead to the return value of F when compared with the rest of the windows. For window size 1 (S1) we have three predicate functions (A, B, and C) with only one, A, leading to a unique return value with score 0.2. This score is the relative frequency of window AF,[2] when compared with all other windows leading to F, for all return values. We add a score to the total score when a window leads to single return value of F since this situation is the only case that "predicts" a return value. We consider only the smallest windows that lead to a single value (*e.g.,* A is no longer considered for S2 and KB, LB, MB for S3) because larger windows do not add anything to our knowledge for the return value.

3.3 Caveats and Limitations

Since our tools operate at the application level, they do not provide supervision or control for any operating system kernel code. We plan to combine our tool with system call interposition or virtual-machine monitors to protect the kernel. The failure semantics and corresponding return values of system calls are largely fixed (at least within the same OS) and thus better understood.

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 (SEV) of shaping the behavior of the system towards an acceptable profile of failure dynamics, but SEV is still not optimal.

Automating a response strategy is difficult, as it is often unclear what to do in response to an error or attack. A response system must anticipate the intent of the programmer, even if that intent was not well expressed. In order to have a reasonable basis for evaluating self-healing mechanisms, a general metric of system correctness needs to be constructed. Most approaches to self-healing cause semantic incorrectness by executing through a fault. This scale of behavior ranges from coarsely correct toward a very fine-grained notion of correct program execution: (a) not crashing the application, (b) not raising any externally noticeable events that were not previously visible, (c) execution behavior becomes correct with respect to the programmer's implementation, (d) execution behavior becomes correct with respect to the programmer's intent, (e) correct with respect to the programmer's understanding, and (f) absolute correctness. Note that this last case is super-optimal – a programmer's mistake may be fixed.

Finally, the application profile is highly binary-dependent. Thus, our system must recognize when an older profile is no longer applicable (and a new one needs to be built), *e.g.,* as a result of a new version of the application being rolled out, or due to the application of a patch. There are several ways for detecting this, not least the modified time of the program image on the disk.

4 Implementation

Lugrind is a binary-level application behavior profiler. It is implemented as a tool for the open-source Valgrind (`v3.1.0`) binary instrumentation framework, which enables plug-ins to inject instrumentation into a running application. Lugrind is based on Josef Weidendorfer's Callgrind[1] tool, and its primary job is to maintain a collection of return values for each function instance.

4.1 Basic Design

Binary-level function profiling is more difficult than may initially be expected. Functions are source level artifacts that have only rough analogues at the machine level. Since source-level notions of a function may be arbitrarily altered by the compiler, or control flow may be interrupted by signal handling, it is difficult to cover all cases of function entry and exit. The Callgrind tool does a remarkable job of tracking function entry and exit in most cases; Lugrind augments Callgrind with additional control for recording and replacing information and handling the cases where Callgrind is not able to produce a valid execution trace. We also incorporate small changes to Valgrind itself to expose the architectural-level information Lugrind requires, since only an intermediate instruction representation (and not the original instruction stream) is available to plug-ins.

Lugrind needs to accomplish a few basic tasks, including recording function entrance events, exit events, and return values. It also needs to build a profile by calculating the conditional probability [28, 3, 7] between function sequences and return values. Finally, it must schedule appropriate return values upon detection of a deviation in behavior (*i.e.,* a fault or exploit).

The system operates mainly in a mode that combines detection and self-healing. Offline training is also supported – the system provides the option to record a list of return values for each function instance 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 has to be restarted. If a fault manifests in a particular function, Lugrind then returns from that function with a return value selected from the collection of values for this function, given the appropriate calling context supplied by the features described in Section 3. Future calls to that function can be "skipped" if Lugrind replaces the call to the function with an immediate return of the SEV value. We term the selection of SEV values "return value scheduling."

4.2 Return Value Selection

Return value scheduling can use a number of selection strategies. The simplest is to choose the first observed return value for a function instance. Second, a round-robin strategy can iterate through all observed return values. If new values are inserted into this set, already-used values may experience "starvation." To avoid this problem, we can employ the "delayed round-robin" variation to completely iterate through the current set before new values are considered.

In general, return values can be rank-ordered by some priority. One priority metric is to match the current function instance with the closest instance seen during training. This ranking method requires that Lugrind also maintain a suitable amount of context for each function instance in order to uniquely identify it. This context is based partly on the current state of the call stack as well as previous sibling functions. We discussed the features of this context in Section 3.1. As a small example of this type of selection strategy, if function `c()` always returns the value "-1" when functions `a()` and `b()` proceed it, and this configuration

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

matches the context at the moment of healing, then “-1” is a good candidate value. Lugrind calculates the conditional probability of return values for an adjustable width window of this history.

Return value scheduling provides the motivation for continuous evaluation of the execution profile that results from the selection of a particular return value. This automatic repair validation is the third task that intrusion defense systems should accomplish. Lugrind provides such a feedback mechanism to see how well subsequent configurations of the execution profile match the expected configuration given a replacement return value. Keeping track of the behavior of the application *after* return value scheduling assists in both repair validation as well as detection of abnormal behavior due to the continuing existence of a compromise or fault despite the self-healing action.

4.3 Additional Control

In order to evaluate our system, we needed to incorporate a small amount of control functionality into the tool. The main purpose of this functionality is to simulate the injection of errors into the application. Normally, these events would be generated by an array of sensors such as those used by STEM [25] or noticed by Lugrind itself as behavior deviates from the profile.

This control plane can inject errors either probabilistically 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 and represent the execution of that function. Since every function entry point is already instrumented by the tool at runtime, a small portion of that instrumentation is activated to schedule a return value. Besides probabilistic failure of a particular function, the control machinery can be given a message containing the name of a particular function to simulate failure for. In either case, the tool prints out which function fails, some diagnostic information (*e.g.,* how it selected a value), and the value itself.

While it would be optimal (in one sense) for the programmer to inform Lugrind what these values are by annotating the source code of each function, such a workflow is often infeasible. Determining what these values should be is the responsibility of our training phase. Lugrind supports the ability to load a profile at system startup; the profile represents a vestigial policy that can be enforced during runtime. Furthermore, the profile can be supplied automatically, created by the programmer, or written by the system administrator. While the raw data for building a profile can become quite large (on the order of hundreds of megabytes), generated profiles that are loaded by Lugrind during supervision range from a few KB to 1.5 MB.

5 Evaluation

This section demonstrates the value of our approach by using our tool that is able to capture the complete application execution flow at the binary level. The evaluation consists of the following tasks:

1. Generate reliable profiles of application execution behavior. These profiles are based on the binary call graph features combined with the return values of each function instance.

2. Characterize the effectiveness of “smart error virtualization” (SEV) for both interactive and server applications.

3. Perform microbenchmarks to show that the system can be reasonably used for real environments.

We explicitly do not investigate attack detection capabilities (including an analysis of false positive/false negative rates) in the paper’s current form. This evaluation is part of ongoing work, and we expect to be

Table 1: *Number of Unique Functions for Each Application.*

Application	# of Functions	Application	# of Functions
xterm (X.Org 6.7.0)	2111	gcc (GNU v3.4.4)	294
md5sum	239	wget (GNU v1.10.2)	846
ssh (OpenSSH 3.9p1)	1362	httpd (Apache/2.0.53)	1123
bigo	129		

able to report on it shortly. These experiments seek to identify how quickly either control or data flow deviates from the learned profile for both real attacks and artificially injected faults. As an anecdotal account of the latter, we force `_dl_cache_libcmp` to return a value of zero for our tested applications. Doing so drastically changes the behavior of the application; instead of a couple of calls to this function (that presumably ends with a valid lookup or comparison), a long chain of repeated `_dl_cache_libcmp` calls appears in the profile, and the application does not execute "correctly" (recall the discussion in Section 3.3: observed responses range from crashing to incorrect output). Similar behavior can be observed for other functions. While we recount our manual observations of these behavior deviations, the system is built to do so automatically, but we have not yet performed a rigorous evaluation.

5.1 Experimental Setup

For the experiments dealing with profile creation, we processed the output of Lugrind's training phase on multiple runs of a wide range of real and specially crafted applications. We test and analyze applications that are representative of the software that runs on current server and desktop Unix environments. Due to space limitations, we present the results for a subset of these applications. We refer the interested reader to our web site[2] for a more extensive collection of data. We consider the following: xterm (X.Org 6.7.0), gcc (GNU v3.4.4), md5sum, wget (GNU v1.10.2), ssh (OpenSSH 3.9p1) and httpd (Apache/2.0.53). In addition, we use some crafted test applications to verify that both the data and the methods used to process them are correct. In our study, we include only one of these applications (bigo) 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 displayed in Table 1.

5.2 Performance Microbenchmarks

The goal of this paper is to demonstrate the merit of binary-level application profiling, and most of this section is dedicated to showing that this profiling is feasible and worthwhile. Independent of the efficacy claims, however, we first show that Lugrind's performance impact is not prohibitive. We employ the same set of applications listed in Table 1. All results are in seconds, except for the latter set of `httpd` results, which is presented in requests per second. There are four sets of data: `native` (running the application directly on the host), `Nulgrind` (just binary translation without instrumentation), `Lugrind-T` (instrumentation in training mode), and `Lugrind-R` (instrumentation in detection mode). The medians were collected from 10 runs of each application and measurements were fairly consistent; standard deviations ranged from 0.001 to 0.683. The `xterm` tests were run over an SSH tunnel on a local network. The `md5sum` test was run on a 1.8KB `/etc/passwd` file. The `xterm`, `sh`, and `ssh` tests were run with the command '`exit`' as an

[2] `http://anonymized`

Table 2: *Performance Impact of Binary-Level Profiling.* The impact of binary-level instrumentation on a variety of applications is shown through the median execution times for those applications in four scenarios. Except for the latter set of `httpd` results, which are in requests per second, measurements are in seconds. Note that Lugrind-R has a lower `httpd` timing measurement than Nulgrind – this discrepancy, while unexpected, is not significant and is probably due to either or both (a) client load or (b) network conditions.

	xterm	**sh**	**md5sum**	**wget**	**gcc**	**bigo**	**ssh**	**httpd**	**httpd (req/s)**
native	1.083	0.003	0.002	0.150	0.063	0.002	0.293	3.49	36.43
Nulgrind	3.718	0.440	0.292	0.840	0.485	0.218	1.473	9.59	14.24
Lugrind-R	12.677	0.596	0.365	1.121	0.598	0.266	2.18	9.52	13.39
Lugrind-T	63.10	0.727	0.421	1.663	0.731	0.307	7.399	87.89	1.42

argument. For `ssh`, an authentication agent was used to eliminate human reaction time. Finally, the `httpd` tests were performed over a local network with the Apache `flood` tool, and the timing measurements are from the perspective of the client. We see that the training phase of Lugrind, as expected, incurs a hefty performance penalty, but the detection phase is fairly lightweight – it does not add much beyond the cost of binary translation and emulation (shown in the `Nulgrind` results). Although the cost is high, we believe it is a reasonable starting point from which to begin optimizing performance using a variety of software and hardware-based techniques that we intend to investigate in future work.

5.3 Profile Generation & Effectiveness of SEV

We will begin by assessing our potential to efficiently generate profiles that in turn predict the return values of functions. Figure 4 provides some insight into the feasibility of this analysis. 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 because there would be too many valid windows. Small variations of control flow would have a high probability of being legal – exactly the opposite conditions desirable for detection.

We cannot "repair" a function without examining the range of return values that it produces. In addition, we would like to determine a function's return value well in advance based on the execution context. As Section 3.2 explains, the notion of return value "predictability" is defined as a value from 0..1 for a specific context window size (*i.e.,* the history of execution). A predictability value of 1 means that for a given function and context window size, we can fully predict the function's return value. Figure 3 shows the average predictability for the set of examined applications. This figure 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's size and complexity are significantly higher than the rest of the 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 6 and Figure 5 we present results for different runs of `httpd` (Apache) and `wget`. The `wget` utility was executed using different command line arguments and

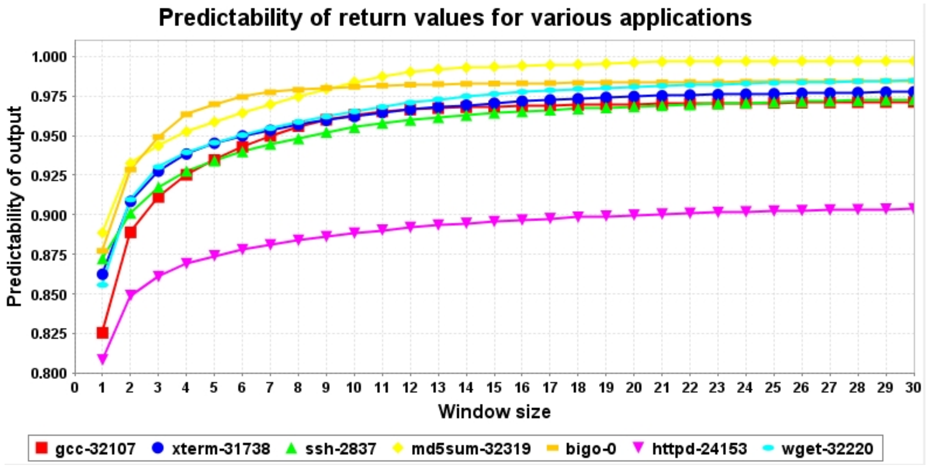

Figure 3: *Average Predictability of Return Values.* This graph shows our ability to predict the return value for various applications against a variable context window size (*i.e.,* the history of execution). Window sizes of more than 15 achieve an average prediction of more than 97% for all applications other than `httpd`, which still has a prediction rate of more than 90%.

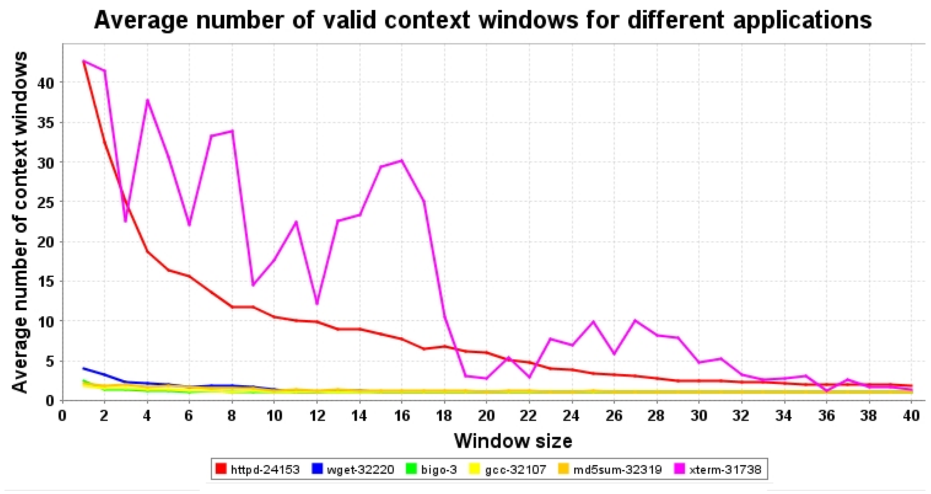

Figure 4: *Drop in Average Valid Window Context.* This graph shows that the amount of unique execution contexts we need to store to detect changes in control flow decreases as window size increases. `xterm` is a special case because it executes a number of other applications. If we consider the ratio of valid context windows to all possible permutations of functions, then we would see an even sharper decrease.

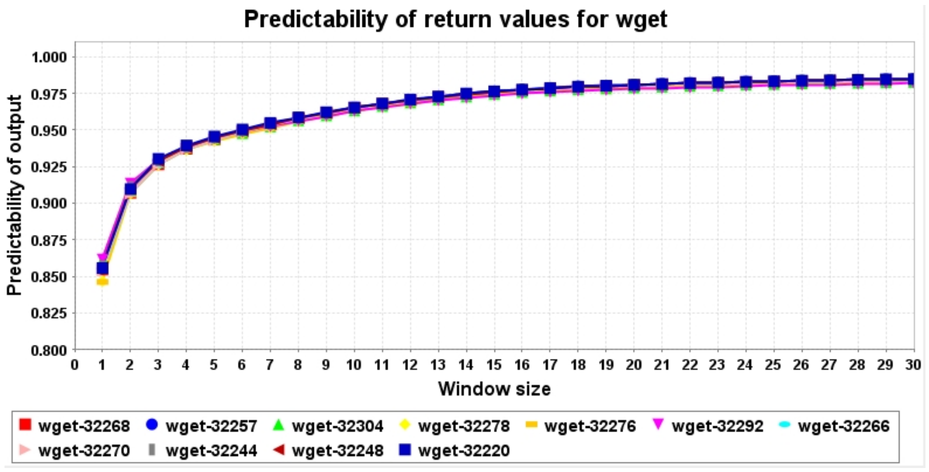

Figure 5: *Average Predictability of Return Values for different runs of wget*. Although there are 11 runs for `wget`, each individual run is both highly predictable (>98%) and very similar to the other's behavior for different window sizes.

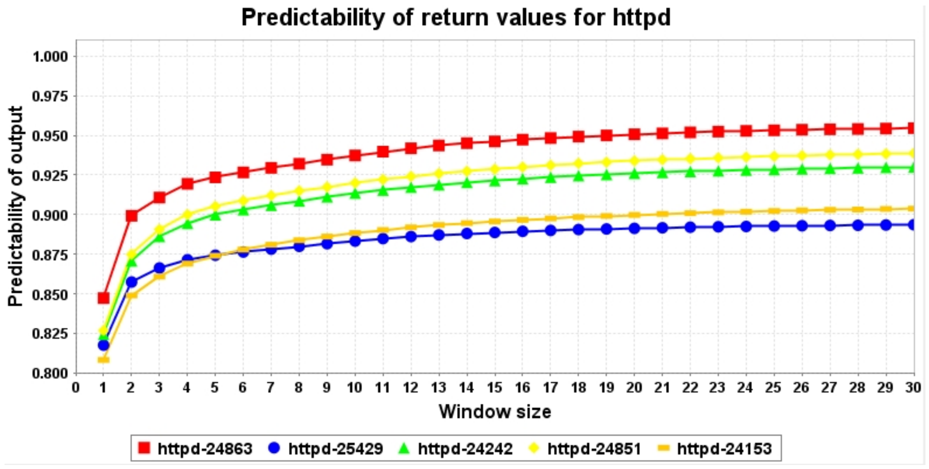

Figure 6: *Average Predictability of Return Values for httpd and for different runs*. Although return value prediction remains high (>90%) for `httpd`, some variations are observable between the different runs. This phenomena is encouraging because it suggests that the profile can be specialized to an application's use at a particular site.

target sites. Apache was used as a daemon with the same configuration file but exposed to a different set of requests for each of the runs. As we expected, `wget` has almost identical behavior between different runs: both the function call graph and the generated return values are almost identical. On the other hand, Apache has runs which appear to have small but noticeable differences. However, as reflected in the average plots, all of the runs still have high predictability. A few questions remain to be answered. How effective is our method in predicting the return values of individual functions? Are there functions that we cannot predict well? How many functions are there of this type? What are the reasons we fail to predict their return values?

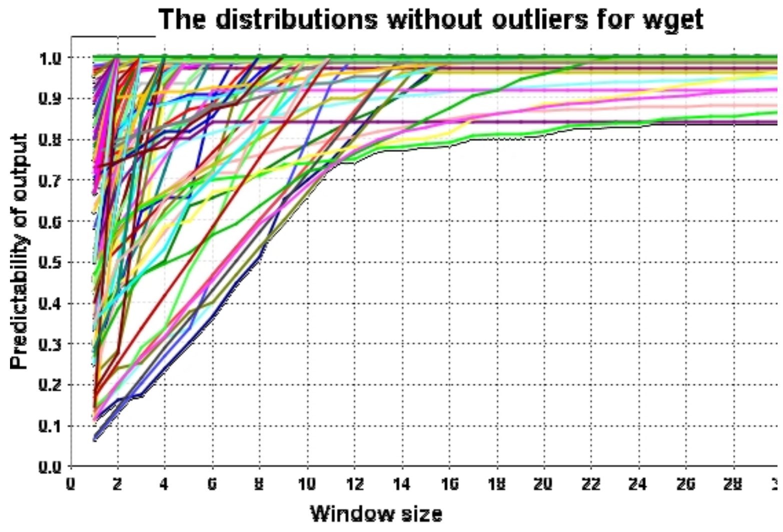

Figure 7: *Predictability of Return Values for wget functions*. Each line represents a unique function and its predictability evolution as context window size increases. Most functions stabilize to high predictability values after a window size of 10. This graph does not include a small set 9 of outlier functions (functions that are 2 standard deviations from the average).

To answer the previous questions we need to take an even closer look at the measurements we have for the individual function predictability. As shown in Table 1, the applications we examine have a considerable number of functions. To present meaningful results we decided to remove functions from our prediction graphs that have a prediction of two standard deviations or greater from the average. The evolution of predictability for `wget` and `httpd` is illustrated in Figure 7 and Figure 8. We can clearly observe what was apparent from the plots of the averages: the majority of the functions are fully predictable – and for small context windows! However, a small percentage of the functions produce return values which are highly unpredictable. We claim that 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 function is a simple example. In practice, as we can deduce from our experiments (see Figure 9), the number of such “outlier” functions is rather small in comparison to the total number of well–behaved, predictable functions.

6 Summary and Future Work

We propose a novel approach to profiling application behavior, modeling the return values of internal functions during runtime. Our scheme captures aspects of both control and data flow. Our system prototype,

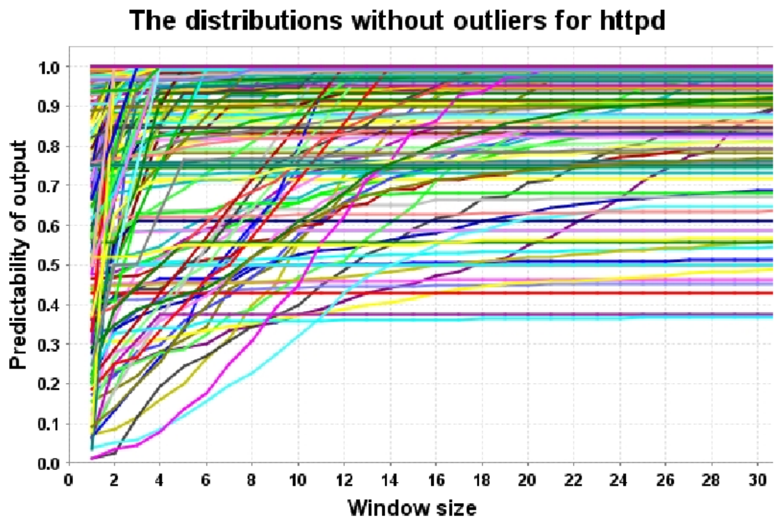

Figure 8: *Predictability of Return Values for httpd functions*. Each line represents a unique function and its predictability evolution as context window size increases. As expected, `httpd` has more functions that diverge from the average. In addition, it has more outliers, as shown by Figure 9.

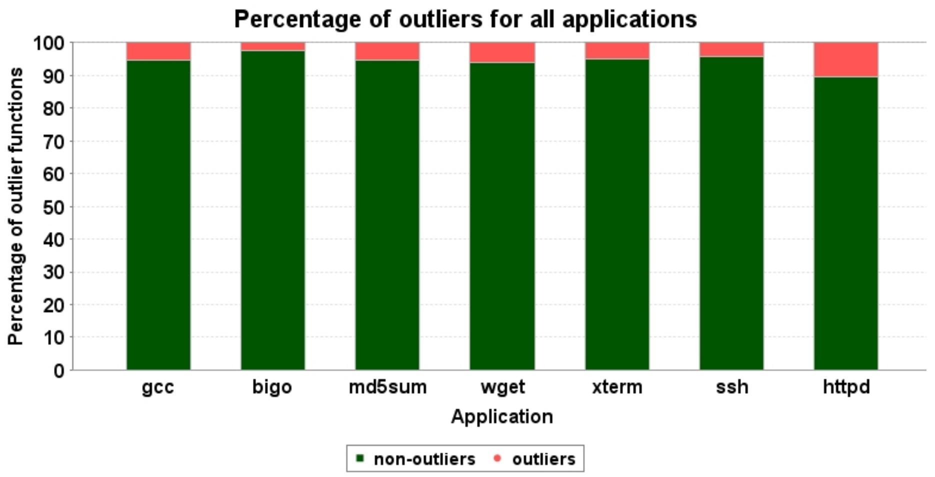

Figure 9: *Percentage of Outliers*. Each bar is split into two parts: the upper represents the percentage of the functions characterized as "outliers" while the lower contains the rest. "Outliers" are functions which deviate from the average by at least two standard deviations for all windows of size more than 10.

Lugrind, builds these profiles of application execution behavior based on information learned from unmodified binaries at runtime. No application, library, compiler, or OS-level functionality that needs to be built in or linked against. We experimentally showed that by using a window of return values that includes the values returned by sibling and parent functions, we can predict the return value of any given function with a 97% probability. The information contained in the profile can be concurrently leveraged by our system for the three major intrusion defense activities: detection, repair, and repair validation.

We have a number of interesting directions for future work, ranging from using PIN [15] to expanding the return value profile to include type information. In addition, the profiles for most applications include a roughly common preamble that deals with loading the libraries needed by the application. We plan to examine how much impact this preamble has on the proportion of "common" functions listed in Appendix Figure 11 as a way to reduce the profile size and improve performance.

References

[1] Bhatkar, S., Chaturvedi, A., and Sekar., R. Dataflow Anomaly Detection. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).

[2] Buck, B., and Hollingsworth, J. K. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications 14*, 4 (Winter 2000), 317–329.

[3] Bui, L., Hershkop, S., and Stolfo, S. Unsupervised Anomaly Detection in Computer Security and an Application to File System Access. In *Proceedings of ISMIS* (2005).

[4] Chari, S. N., and 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).

[5] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the* 14^{th} *USENIX Security Symposium* (August 2005), pp. 177–191.

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

[7] Eskin, E., Lee, W., and 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).

[8] Feng, H. H., Kolesnikov, O., Fogla, P., Lee, W., and Gong, W. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (May 2003).

[9] Gao, D., Reiter, M. K., and Song, D. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2004).

[10] Gao, D., Reiter, M. K., and Song, D. Behavioral Distance for Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 63–81.

[11] Giffin, J. T., Dagon, D., Jha, S., Lee, W., and Miller, B. P. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005).

[12] Hofmeyr, S. A., Somayaji, A., and Forrest, S. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[13] Lam, L. C., and 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).

[14] LIANG, Z., AND SEKAR, R. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[15] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD., K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)* (June 2005).

[16] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science* (2003), vol. 89.

[17] NEWSOME, J., BRUMLEY, D., AND SONG, D. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)* (February 2006).

[18] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Annual Network and Distributed System Security Symposium* (February 2005).

[19] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2002)* (December 2002), pp. 361–376.

[20] PROVOS, N. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium* (August 2003), pp. 207–225.

[21] QIN, F., TUCEK, J., SUNDARESAN, J., AND 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)* (2005).

[22] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., AND LEU, T. 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).

[23] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND 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).

[24] SIDIROGLOU, S., LAADAN, O., NIEH, J., AND KEROMYTIS, A. ASSURE: Autonomic Software Self-Healing Using Error Virtualization Rescue Points. In submission to OSDI 2006, 2006.

[25] SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference* (April 2005), pp. 149–161.

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

[27] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium* (August 2000).

[28] STOLFO, S. J., APAP, F., ESKIN, E., HELLER, K., HERSHKOP, S., HONIG, A., AND SVORE, K. A Comparative Evaluation of Two Algorithms for Windows Registry Anomaly Detection. *Journal of Computer Security 13*, 4 (2005).

[29] WAGNER, D., AND SOTO, P. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2002).

[30] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM* (August 2004).

[31] Xu, J., Ning, P., Kil, C., Zhai, Y., and Bookholt, C. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS)* (November 2005).

A Appendix: Additional Graphs

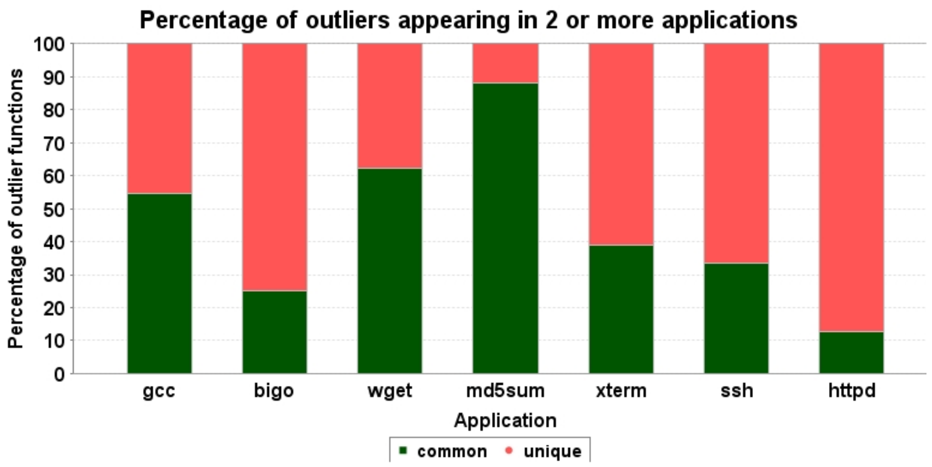

Figure 10: *Outliers appearing in at least two applications.* Each bar represents the totality of the outliers spliting them into two groups: "common" outliers that appear as outliers in two or more applications and the rest. We can clearly see that all applications have common outliers and some of them more than others upto 90%

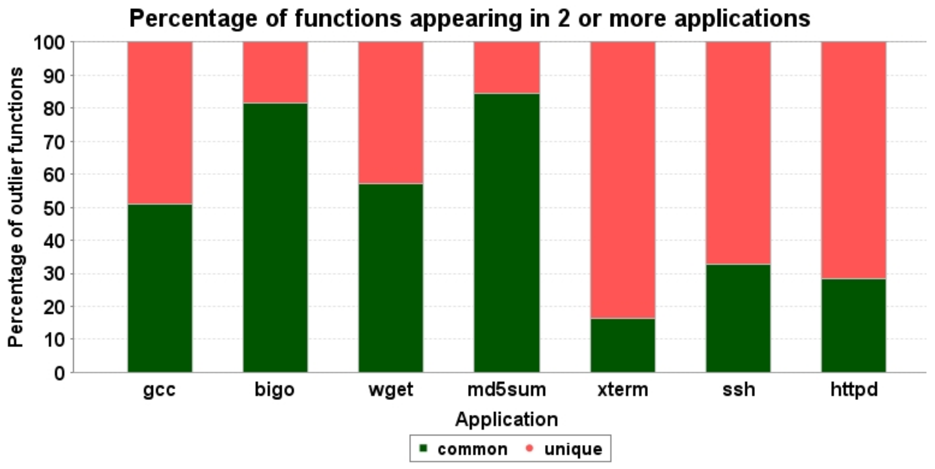

Figure 11: *Functions appearing in at least two applications*. Each bar represents all functions for an application. We separate the functions into two groups: functions that appear in two or more applications which we call "common" and the rest. Again, for most applications, there is a significant fraction of functions that are "common" making the predictability of the return values easier.