Isolation of Malicious External Inputs in a Security Focused Adaptive Execution Environment

Aaron Paulos, Partha Pal, Richard Schantz, Brett Benyo
BBN Technologies
Cambridge, USA
{apaulos, ppal, schantz, bbenyo}@bbn.com

David Johnson, Mike Hibler, Eric Eide
University of Utah
Salt Lake City, USA
{johnsond, hibler, eeide}@cs.utah.edu

***Abstract*— Reliable isolation of malicious application inputs is necessary for preventing the future success of an observed novel attack after the initial incident. In this paper we describe, measure and analyze, Input-Reduction, a technique that can quickly isolate malicious external inputs that embody unforeseen and potentially novel attacks, from other benign application inputs. The Input-Reduction technique is integrated into an advanced, security-focused, and adaptive execution environment that automates diagnosis and repair. In experiments we show that Input-Reduction is highly accurate and efficient in isolating attack inputs and determining casual relations between inputs. We also measure and show that the cost incurred by key services that support reliable reproduction and fast attack isolation is reasonable in the adaptive execution environment.**

Keywords*— *Adaptive Security; Execution Environment; Novel Attacks; Record & Replay; Survivability;

I. INTRODUCTION

Considering the recent escalation in sophisticated attacks against high-value systems, we believe there is a fundamental need for security technologies to *adaptively respond* to compromises stemming from completely unforeseen attacks that exploit previously unknown vulnerabilities. Many procedures for dealing with the aftermath of novel attacks are manual, time consuming and require a high-level of cyber-security and system's expertise. Example corrective actions include cleansing a system of rogue code, repairing damaged data, patching faulty code and defenses, restoring trust in existing credentials and restoring service availability. While some of these corrective actions can be automated, repair actions usually depend on a significant amount of accurate knowledge about the how the attack was delivered, and what happened between the time the attack was executed and the time it was detected. In this context, one of the main difficulties encountered in the aftermath of a compromise is to reliably reproduce the undesired application state or failure with a minimal set of application inputs. Such a technique when coupled with an adaptive repair toolkit will help facilitate and automate a number of corrective actions in a timely manner.

In this paper, we describe an on-the-fly dynamic search technique called Input-Reduction, as a method for isolating fault-triggering inputs delivered to a protected application. Input-Reduction decouples the detection of undesired conditions (i.e., fault manifestations) from isolation decisions. This allows Input-Reduction to integrate with a wide range of sensors capable of detecting undesired conditions, failure states and underlying faults.

This paper also describes and evaluates how we integrate Input-Reduction with the prototype *Advanced Adaptive Applications (A3)* execution environment. In A3, the reliable reproduction of an observed undesired condition is a critical step towards reasoning about the undesired manifestation as well as any underlying security problems that may exist. Isolating attack inputs enables A3 to conduct security-focused hypothesis testing of the application's configuration, underlying binaries and security policies more efficiently. The goal of such hypothesis testing is to discover a corrective action that makes the application more resilient against the initial attack.

In this paper, we make the following three contributions:

- We describe Input-Reduction, a technique for isolating malicious inputs from benign inputs.
- We present an integration and evaluation of Input-Reduction within a secure and adaptive execution environment.
- We evaluate and decompose the cost of infrastructure services in A3 that facilitate Input-Reduction.

The rest of the paper is organized as follows. In Section II we describe the A3 prototype within which Input-Reduction is implemented and evaluated. Section III describes the Input-Reduction procedure. Section IV presents an experimental evaluation of Input-Reduction. We further examine the integrated cost implications of services and capabilities within A3 that support recording and detection of undesired conditions, which Input-Reduction uses. Section V discusses and contrasts the goals and features of Input-Reduction and A3 with other related works. Section VI concludes the paper with a brief discussion of future work and our contributions.

This work is being supported by the Unites States Air Force and DARPA under Contract No. FA8750-10-C-0242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

978-0-7695-5008-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ARES.2013.15

II. THE A3 ADAPTIVE EXECUTION ENVIRONMENT

Input-Reduction is realized in the context of the prototype A3 survivability-focused execution environment [1, 2]. A3 containerizes individual applications, enforces mandatory mediation of application input/output (I/O), and provides a framework for developing adaptive middleware procedures to dynamically counter attacks. An earlier work [1] described and evaluated an initial implementation of the preventive features of A3 based upon mandatory mediation of application inputs in *crumple zones,* a software structure that absorbs attacks. Later, [2] described the initial design and preliminary evaluation of a key subset of an envisioned portfolio of security-focused adaptive procedures that A3 supports. That work postulated that the performance of adaptive procedures for attack diagnostics and hypothesis-driven patch testing would greatly benefit from efficient isolation and reduction of inputs.

To isolate malicious inputs, Input-Reduction uses A3's *replay services* to load a check pointed state and adaptively play-back a recorded execution. Input-Reduction also uses A3's capabilities to detect undesirable application and system states, for the purpose of guiding its search. These integrated capabilities demand a level of sophisticated execution management that is not commonly available when applications are run in current application environments, and are highlighted in the description of A3 in this section.

In A3, a protected application executes in its own container and is given the impression that it has the entire host to itself. In the current prototype, containers are implemented as guest Virtual Machines (VM) running over a Xen [21] microkernel. The protected application's interactions with the network and disk are intercepted by a network crumple zone (NCZ) and a storage crumple zone (SCZ) respectively, each implemented as its own container. The crumple zones are responsible for enforcing preventive policies on interactions through them. The three VMs, containing the protected application and the NCZ and SCZ, are collectively called the *production conglomerate*, and enable the advanced execution management features that are necessary to perform security-focused adaptive procedures like Input-Reduction. Most of the adaptive maneuvers are actually carried out in an *experiment conglomerate,* a separate set of VMs that are independent of the production conglomerate, leaving the production conglomerate free to perform its primary task. When an undesired condition is detected, the application in the production conglomerate is rolled-back to an earlier check pointed state, and an experiment conglomerate is created to perform security-focused adaptive procedures. The A3 Controller is used to manage A3 and orchestrate the experimentation process. Human operators may use a *GUI Dashboard* to manipulate the state of the experiment conglomerate, e.g., select a check-point, or configure, execute and monitor an adaptive procedure. Fig. 1 depicts the A3 environment.

A3 supports replay at two levels: at the application level, where an application's inputs are played back, and at the VM level, where the entire VM's operation is played back deterministically. These two replay options address different

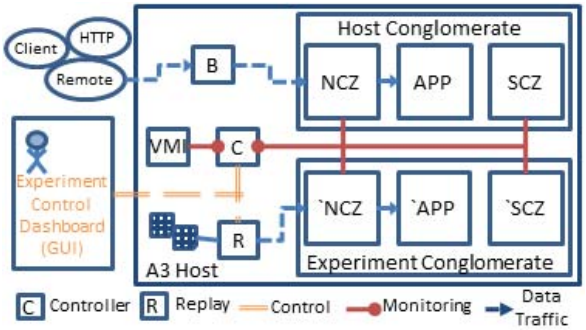

Fig. 1 - A3's Replay Infrastructure

tradeoffs, such as overhead incurred and playback fidelity, the merits of which are beyond the scope of this paper. For Input-Reduction's evaluation of inputs, we currently use the more mature application-level replay. Application replay requires *detailed-enough* recordings that will allow reconstruction and playback of discrete inputs to the application. For Input-Reduction against a server-class application, this amounts to capturing packet-level network events over mediated channels and crumple zones. This type and level of playback is well-suited for revisiting the *input-output behavior* of an application such as a web-server.

A3's mandatory mediation of application I/O provides a suitable interposition point for supporting record and replay of applications that are driven by discrete external inputs. Request-Reply protocols are widely used throughout server-class applications and embody this messaging style paradigm. To evaluate Input-Reduction, we use a file-storage web-service, implemented with Apache and PHP. To record network inputs with a sufficient level of detail to play back HTTP requests, we have implemented a service based on libpcap[9], called httpdump. The httpdump recording structure allows replay to totally order HTTP requests in terms of the time they pass the network interface, and to derive an understanding of the requesting clients' session and concurrency semantics, without requiring distributed logging and synchronized clocks. Request semantics can be derived using the client's network of origin, client's authentication credentials, request inter-arrival times and causal dependencies in the web-service requests (e.g., a file F must be uploaded before a download of file F).

In A3, detection of undesirable conditions is provided by preventive policies in crumple zones and VM Introspection (VMI) of each container's execution state. While crumple zones monitor edge interactions with the application, to monitor process execution we implement a VMI service that inspects the machine state of A3 containers from Xen's hypervisor. Detection of undesirables serves two purposes: first, it acts as a trigger to initiate post-incident procedures such as Input-Reduction, and second, during Input-Reduction these detections are used to help guide the search by indicating whether a given replay (i) contributes to reproducing the undesired condition, or (ii) triggers additional watch conditions imposed as part of adaptive diagnostic analyses of an observed attack.

It might seem counter-intuitive that preventive policies can indicate an undesired condition, since enforcement of preventive policies implies that a non-compliant interaction has been blocked. However, A3 crumple zone policies control outbound interactions as well—so even though the policy may prevent the exfiltration attempt, the compromise that led to that manifestation probably occurred in the past. Furthermore there are no guarantees that the policies are perfect, especially in the context of novel attacks. It might be the case that part or all of the attack sequence successfully passed through the crumple zone, installed a backdoor, and then a subsequent interaction violates the policy. In all cases, we assume that an observed violation indicates compromise.

In this work, we implement VMI capabilities as a configurable service called vmprobesys. Vmprobesys is built upon XenAccess [19], a low-level library for accessing the machine state of a guest VM in Xen. Vmprobesys monitors and reports function calls from the guest OS that match a filtering specification involving process identifiers and system call attributes. Events that match the specification are reported to the A3 Controller as a short observation. An abridged VMI probe specification for our example application is shown in Fig. 2 and used in experimentation later in this paper. In this specification, the probe service is used to evaluate system calls that control process creation and network interactions for the protected application. The directive *ProcessListNames* is used to report a process existence status for the specified process names, here httpd and php-cgi. Vmprobesys reports the process ids (PID and PPID) whenever a process is created or destroyed. This capability is mostly used for auditing purposes and higher level analyses looking for anomalies in the list of active instances of processes. The directives *Functions* and *Filter* are used to monitor the functions whose invocation and subsequent results are of interest. Filterable Functions are listed under the Function directive. The Filter directive defines a condition that can be evaluated over a probed function call and it's return values. For instance, the first Filter specifies that the do_exit function call with a SEGV parameter for an httpd process is an event of interest.

Higher-level analyses and subsequent reactions to observations from vmprobesys are performed at the A3 Controller. Performing high-order analyses inside vmprobesys would severely impact the performance of the guest VM, because monitoring guest function calls involves breakpointing the applications execution. Thus, adding complex logic to analyze multiple observations inside the vmprobesys code will adversely impact the applications performance. This design decision is acceptable, because reporting latencies from vmprobesys to A3 controller are consistently less than 10ms.

```
ProcessListNames httpd, php-cgi
Functions sys_execve, sys_waitpid, do_exit, sys_fork,
sys_clone, sys_socketcall
Filter f=do_exit,name=^httpd,arg=code:signal,aval=SEGV
Filter f=sys_execve,name=^httpd
Filter f=sys_socketcall,name=^httpd,arg=call,aval=accept
Filter f=sys_socketcall,name=^httpd,arg=call,aval=connect
Filter f=sys_waitpid,name=^httpd,when=post,rval=^[1-9]
```

Fig. 2 - Abridged VMI Probe Specification

III. INPUT-REDUCTION

Input-Reduction is used to reorganize a recorded *input set I* in such a way that sub-sets of I can be matched against a *user-defined objective function F,* also known as a predicate. The predicate is defined when a replay experimenter, or other automated agent, selects a sub-set of any undesirable observations that were reported within the same interval as the input recording. In most cases, the predicate F will be a single undesirable observation detected by A3's preventive protections, as that observation is usually the one that triggered an adaptation by A3. In other cases, multiple observations might have been reported that warrant individual inspection on a case-by-case basis. After the predicate is selected, Input-Reduction orchestrates four search procedures (called phases) that submit variations of recorded inputs to the application and wait for an observed event(s) that matches the predicate. The inputs that do not contribute to making F true are removed from further consideration. In this way, the search is used to "reduce" a full recording to a *smaller sub-set R* of recorded inputs.

Input-Reduction is designed around two goals and an assumption. It should quickly isolate a reduced input set. It is important to minimize the number of state resets (check-point loading), as these operations tend to be very expensive. Finally, for many types of attacks, the attack payload will be contained within a single application input that was processed right before an observed anomaly is detected. For attacks that lie dormant, the algorithm can further selectively apply longer timeouts, although, this class of attack is not our main focus. Under these guidelines, Input-Reduction will discover if an observation is reproducible in relation to a given starting check-point (state), and, if reproducible, it can uniquely isolate up to three dependent inputs that contribute to the predicate, or reduce the input set to a sub-set range of 4+ dependent inputs. When considering ordered dependent inputs, Input-Reduction aims to identify the first case that triggers the predicate. Thus, it is not optimized for more complex input sets such as symmetric input (i.e., ABCBA where A followed by B triggers the predicate). The complexity of Input-Reduction is approximately linear in the number of input submissions. While the processing times for each individual submission may vary, the complexity of Input-Reduction is always dominated by the trial-setup and state reset costs, so the size of the input does not tend to matter in practice. Furthermore, A3 can manage the input-set size by varying the frequency of check-pointing, thus preventing unbounded growth in input recordings.

Input-Reduction is inherently expressive, fast, and flexible, but provides no optimal search guarantees, as this will unnecessarily slow execution time. That is, the search may quickly reduce an input set to contain the malicious inputs, plus a few other residual benign inputs. While Input-Reduction is generalizable to the reduction of discrete messaging inputs, this work demonstrates a specialized search for HTTP inputs. In A3, we use Input-Reduction to

submit HTTP requests as inputs via the replay service, and implement a predicate interface to handle observations reported from A3's Host Controller as shown in Fig 3. In A3, we use tunable parameters to account for client semantics including inter-arrival time of HTTP requests, exact message time stamps, socket-timeouts, and emulation of client thread-pool size. Input-Reduction can also vary replay speed and an end-of-submission-cycle timeout value that is used to determine how long to wait for a match on the predicate after submitting a final input. In the absence of client-side recording, which would not be enforceable against an arbitrary attacker, we feel there is sufficient flexibility in Input-Reduction's ability to submit inputs and emulate client behaviors when coupled with A3's replay service.

The four phases of Input-Reduction are as follows:

Phase 1 – ***Reverse-order Submission*** works backwards from the last recorded input to the first input. For example given a set {1, 2, 3}, Input-Reduction would submit 3, then 2, then 1 as individual inputs. A terminating condition is defined as a predicate match or the expiration of a configurable duration end-of-cycle timeout that follows the last submitted input. In other words, the phase must end after a preconfigured timeout. Input-Reduction always leads with reverse-order submission based upon the assumption that many attacks will be self-contained within a single input near the end of a recorded set, and that costly state resets are only required for initialization. Phase 1 has the added benefit of assessing if the set contains dependent inputs, i.e., ordering semantics. Phase 1 is always followed by Phase 2.

Phase 2 – ***In-order Submission*** follows the reverse order submission phase, only requires a state resets during initialization, and has the same terminating conditions as Phase 1. If the predicate was not triggered in Phase 1, input submission will start from the first recorded input. Otherwise, the submission will start from the last submitted input that directly preceded the predicate match in Phase 1. A Phase 2 predicate match that follows a reverse-order miss in Phase 1 implies that the input set was ordered, which is often useful in future debugging sessions. If a single submitted input triggers the predicate, Input-Reduction is complete and returns a reduced set containing the single input. If the predicate was matched and there is more than one input remaining, Input-Reduction will proceed to Phase 3 if the set is ordered and to Phase 4 if the set is unordered. If

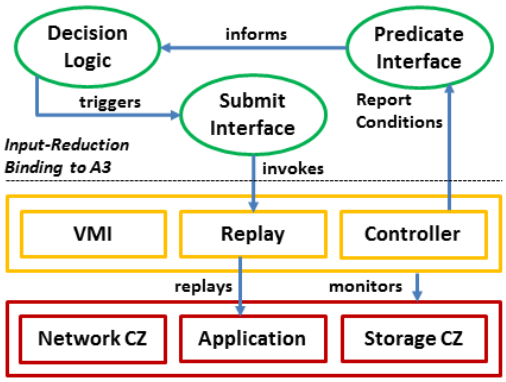

Fig. 3 - Integrating Input-Reduction in A3

the predicate is not matched after the end-of-cycle timeout in Phase 2, the recording may not be capable of reproducing the undesired condition from the starting checkpoint and recorded inputs. This may be due to the level of the recording, the configuration of Input-Reduction's tunable parameters, or the starting checkpoint. In this case, Input-Reduction will signal a negative result and terminate.

Phase 3 – ***Divide, Conquer, Expand (D/C/E)*** takes an *ordered input-set* and proceeds to use a ½ divide, conquer and ½ expand approach to iteratively test smaller, followed by larger, sets and narrow the attack sub-set I. While the D/C/E phase is much more efficient in cumulative input reduction, each pass requires a state reset operation. This is why we do not start with this phase. The main purpose of Phase 3 is to *pre-process ordered sets* that may contain many innocuous inputs to the left of the last input in the set. Phase 3 attempts to eliminate those inputs prior to proceeding to Phase 4 as an efficiency step. The termination case for Phase 3 is defined as the last predicate match that occurs between a *divide iteration* and an *expand iteration*. It occurs when the starting point for successive divide and expand steps is within 1 input and a true-positive against the predicate is observed. Regardless of outcome, Phase 3 invokes Phase 4.

Phase 4 – ***Bookend + D/C/E*** is used to uniquely identify 2 or 3 dependent inputs, and in the worst case, reduce the input set to a smaller range of 4+ inputs from the initial set. Phase 4 takes a reduced input set from Phase 3 or Phase 2 that may contain zero or more innocuous inputs in the middle of that set. Initially, Phase 4 tests that only the first and last input in the set is part of the attack to check for exactly 2 dependent inputs (thus the name bookend). A positive result will occur if Phase 1 and Phase 2 have effectively restricted the input sets two bookends. Phase 4 will try the initial bookend, and if that test fails, it will then use D/C/E over the middle of the set and wrap each D/C/E iteration with input 0 and input N-1 of the phases' starting set. Intuitively, Phase 4 will submit the start of the bookend, then play a step of D/C/E, and then finally submit the last input of the bookend. This will continue until the D/C/E tests, wrapped in the bookend, are complete. As a result of the final phase, Input-Reduction will identify up to three unique inputs, or the bookend plus some range of inputs that contain a mix of malicious and innocuous inputs.

For robustness, each phase includes a parameter to repeatedly retry up-to-N attempts to match the predicate. In terms of minimizing costly state resets, the reverse- and in-order replays only require a single state reset, while the D/C/E and bookend tests require a restart after each division. Since Input-Reduction does not attempt to optimally isolate reduced sets nor takes into consideration time-depended predicates (e.g., hibernating attack), a *false-positive* would be defined as a resulting reduced set that will no longer trigger the predicate effect given an infinite timeout. A *false-negative* would be defined as an irreproducible predicate assuming an accurate starting state, timeout configuration and playback fidelity. It is our intuition that using deterministic-replay for input submission would minimize the chances of false-positives and false-negatives, which we plan to examine in later work.

IV. EVALUATION

To evaluate Input-Reduction we use a 2.2 Ghz quad-core processor, 8GB of memory, and 1CPU and 512MB Xen PV-guests for the A3 containers. As an exemplar protected application, we use a file-storage web-service application implemented with Apache 2.2.19 and PHP 5.3.6. The web-service provides a Wiki document management interface supporting file upload, download, delete and indexing. Web-service clients are physically located on a separate machine connected via a 100Mb LAN at the time of recording. By design, the web-service's implementation and the deployment have been left with embedded security holes, such as misconfigured discretionary access control settings, and function calls which make use of user inputs without input validation that may be exploited remotely. Collectively, these security holes represent many of the CWE/SANS Top 25 Most Dangerous Software Errors [12] and expose the application to a range of common attacks.

In the following three sub-sections, we first examine the efficacy and efficiency of Input-Reduction in isolating malicious inputs. We then examine the fixed costs for detecting undesired conditions in a containerized environment that ultimately drive Input-Reduction. In the final Section IV.C, we examine the VMI costs that are associated with a deeper inspection during Input-Reduction.

A. *Efficacy and Efficiency of Input-Reduction*

To demonstrate and assess the accuracy and performance of Input-Reduction we experiment with two attack recordings that represent common attacks that can be easily packaged in HTTP requests:

Segv – In CVE-2012-0021 [6], a malformed HTTP message with malformed cookie field may be used to trigger a segmentation-fault in an Apache process. The attack's payload is contained within a single HTTP request, and the segfault signal is almost immediate. This undesired condition is easily detected by vmprobesys' monitoring of SIGSEGV exits as described in Section II.

PHP Backdoor – The Syrian-Shell [22] PHP backdoor is an application-level attack. The attack is loaded into Apache's htdocs directory via a misconfigured Discretionary Access Control setting and a path-traversal vulnerability that is exploited during a file storage operation. The detection of a policy violation occurs after two dependent requests: an HTTP POST operation to upload the Syrian Shell into the htdocs directory, then a HTTP GET invocation on the attack script. The second request triggers a bind call when attempting to open a back door. This bind call is also detected by vmprobesys.

With A3, Input-Reduction can be configured to use different playback speeds, including "as recorded" using the same inter-arrival time between requests that was observed during recording, and "fixed-time", where requests are played back at a chosen fixed interval. To illustrate Input-Reduction, Fig. 4 graphs a replay for the PHP backdoor example using recorded time. The time of the replay is plotted on the X-axis in milliseconds, and the y-axis is categorical where each entry represents the submission of a single replay input. Although it is not visible in the

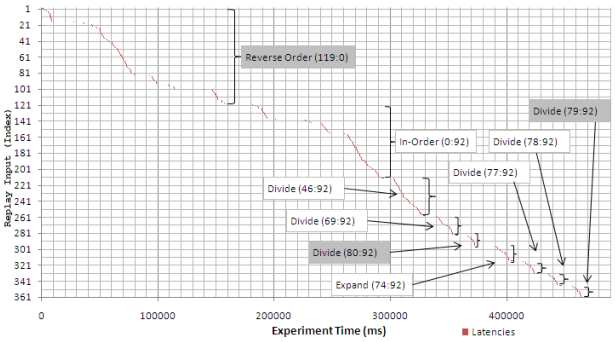

Fig. 4 - Input-Reduction of PHP Backdoor

granularity of the graph, each single input is plotted horizontally, as a column, and conveys the replay's request-to-response latency for each input. Fig. 4 labels the phases of Input-Reduction in colored boxes, where the shaded boxes indicate that the predicate was not triggered in that replay iteration, and white boxes indicate a match.

Since the PHP attack contains two order-dependent inputs, the first phase of the replay, reverse-order, leads to no input reduction. In the second phase, in-order traversal triggers the predicate after the 92nd input. Collectively, these two phases indicate that trigging the predicate is possible given the recorded input set and that the attack is order-dependent. As mentioned earlier, factors including time of the starting checkpoint, log-size, replay-timing, and the existence of causally-related attack inputs will influence predicate reproducibility. For example, if events that occurred prior to the first recorded input of the recorded set are needed to trigger the predicate, then the replay experiment will fail at this point. For this particular attack set, Input-Reduction has now determined that the input-set was indeed ordered, and can proceed to divide, conquer and expand to test for further reductions. From about the 300 second mark until the final divide test, Input-Reduction will reduce the set by 98% before returning the result of the search. The total replay-time using the recorded inter-arrival times of the input set is ~357 seconds. During this particular replay, there were eleven total state resets (application restarts) in the experiment. In absolute terms, the final result of Input-Reduction shown in Fig. 4 actually reduced the PHP Backdoor attack set to two inputs: the POST request which injects the script and the GET request that triggered VMI.

To examine the effectiveness of different Input-Reduction configurations we conduct four unique Input-Reduction experiments in A3:

- ***Input-Reduction Experiment 1*** – With only an experiment conglomerate, we test the PHP backdoor attack set. We execute the replay using the recorded request arrival times, a 3000ms socket-timeout and a 6-client thread-pool.
- ***Input-Reduction Experiment 2*** – Starting from the configuration in experiment 1, we alter the replay speed to be fixed at 333ms between HTTP requests.
- ***Input-Reduction Experiment 3*** – With only an experiment conglomerate, we test the Segv attack set. We execute the replay using a fixed replay

speed of 333ms between requests, a 3000ms socket-timeout and a 6-client thread-pool.

- ***Input-Reduction Experiment 4*** – In a configuration with both a production and experiment conglomerate running on the same host, we run Input-Reduction against the Segv attack set, while at the same time, the production conglomerate processes index requests from 7-clients at 333hz. The replay speed is configured for a fixed 333ms between requests, with the same socket-timeout and client thread pool size as the other experiments.

The results in Table 1 examine and group the four logical phases of Input-Reduction vertically, against the four experiment configurations described above (horizontally). For each phase, we've captured the number of inputs, the size of the reduced set at completion of the phase, the cumulative reduction in set size, the number of passes over the data where each pass indicates that a state reset is required, and excluding the state reset duration(s), the time it took to replay inputs and wait for any observations.

Comparing the results of the PHP backdoor experiments (Exp. 1 and 2), we observe a 98% and 88% input reduction against the initial input set. The additional input reduction accuracy for the recorded-time replay comes at a total cost of 356.91 seconds and one less state reset, as compared to 127.45 seconds for the fixed-time experiment. We derive total time here, by summing the replay times of all four phases. These findings suggest that reasonable input reduction can be achieved quite fast by using a faster than recorded time fixed timing interval for replay. To enhance the accuracy of the replay, slower timing, or in this case, recorded time, can be used on the resulting sets to improve the reduction accuracy even further.

In the first of the two segv experiments (Exp. 3), Input-Reduction identifies the exact attack input (id 148) in 5.66 seconds during two replay phases. Like the earlier PHP Input-Reduction experiments, the first segv experiment does not include a batch of clients concurrently making requests against the protected application in the host conglomerate.

In the second segv experiment (Exp. 4), we rerun the same segv Input-Reduction experiment concurrently with an active production conglomerate that services requests from seven clients at 333Hz. Without question, the additional load impacts Input-Reduction. While the reduction results are quite good, with an input reduction of 92% in 17.48 seconds and two additional state resets, we see a relationship between the additional load on the production conglomerate and the accuracy of Input-Reduction. Since, as evaluators, we know that the attack input is contained in the single non-dependent input 148, when we consider the data in Table 1 we see that the initial reverse order replay overran the attack input by seven inputs, the in-order replay then overran the input by five inputs, and ultimately, the bookend tests failed two consecutive times to end the experiment. Each of the outcomes can be attributed to the timing of the Input-Reduction's test-observe-decide cycle. We believe that the additional resource load (i.e., resource usage) on the experimental platform and the increased volume of observations reported to the controller from the concurrent production conglomerate have the effect of increasing latencies in the test-observe-decide loop of Input-Reduction. In effect, the increased cost leads to timing overruns (i.e., a type of race condition) where the actionable information arrives after the end-of-cycle timeout. One way to redress this effect, would be to increase the end-of-cycle timeout configuration, either automatically (e.g., via exponential decay) or manually on a case-by-case basis, to cope with the increased load from the production conglomerate.

Taken as a whole, the analysis of the Input-Reduction in the context of the HTTP-based web-service application demonstrates that the Input-Reduction is effective and accurate in isolating malicious inputs that reproduce an observed undesired condition at run-time. Both the accuracy and efficiency of Input-Reduction can be affected by configuration of the replay experiment attributes. Attributes such as latencies of observations delivered to the controller, end-of-cycle timeouts, and replay speed all effect outcomes for Input-Reduction. We observe that slower replay speeds generally lead to a greater degree of accuracy in the Input-Reduction outcome. Under this observation, one might be inclined to run Input-Reduction in two or more passes, starting from a fast replay, to rapidly prune innocuous inputs that do not contribute to matching the predicate, and then progress to slowing the replay speed over the newly reduced

Table 1. Comparison of Set-Reduction Replay Experiments

		Phpbackdoor @rec-time	Phpbackdoor @333	Segv @333	Segv with host traffic @333
Phase: Reverse Order	Input Size (Reduced Size)	120 (120)	120 (120)	165 (17)	165 (24)
	Replay Passes	1	1	1	1
	Cumulative Reduction	0 %	0 %	90 %	85 %
	Replay Duration	162.79 seconds	42.00 seconds	5.63 seconds	8.15 seconds
Phase: In-Order	Input Size (Reduced Size)	120 (93)	120 (94)	17	24 (13)
	Replay Passes	1	1	1	1
	Cumulative Reduction	23 %	22 %	99 %	92 %
	Replay Duration	109.79 seconds	32.67 seconds	.03 seconds	4.42 seconds
Phase: Divide, Conquer, Expand	Input Size (Reduced Size)	93 (15)	94 (16)	N/A	N/A
	Replay Passes	7	6		
	Cumulative Reduction	88 %	88 %		
	Replay Duration	83.44 seconds	46.96 seconds		
Phase: Bookend + DCE	Input Size (Reduced Size)	5 (2)	16 (16)	N/A	13 (13)
	Replay Passes	2	2		2
	Cumulative Reduction	98 %	88 %		92 %
	Replay Duration	.89 seconds	5.82 seconds		4.91 seconds

set to enhance the accuracy of the outcome.

B. Fixed Cost Component of Input-Reduction

Preventive policies enforced by VMI and mandatory mediation are necessary to support Input-Reduction in A3. While our initial assessment of the benefits of preventive protections [1] is positive, we wanted to make sure that they do not make the prototype execution environment cost-prohibitive for enabling adaptive procedures such as Input-Reduction. With this motivation, this sub-section analyzes the performance costs of A3's preventive protection services that were used to generate the input and observation recordings used in the previous sections experimentation.

To conduct a fair and measured comparison we first examine the impact of preventive protections on external application clients, where two crumple zones are used to enforce I/O mediation, vmprobesys monitors undesired operational conditions, and application-level recording captures data on the network channel. We then remove preventive protections one layer at a time to decompose and attribute cost to individual protections. As a metric of cost, we measure the HTTP round-trip latency for 1, 3, 5 and 7 concurrent clients issuing three requests per second.

We use the following configuration, defined specifically for the protected web-service application, for preventive protections to exercise A3's core components:

- Storage Crumple Zone
  - Filter file system operations on .exe, .dll, .bat, .sh, .so objects
- Network Crumple Zone
  - Filter HTTP headers >128 bytes in length
  - Filter HTTP requests >10 HTTP Headers
  - Filter out-of-range WS query strings
- Virtual Machine Introspection
  - Apply the specification from Section II

In 24 experiments we test six configurations, slowly increasing the quality and depth of preventive protections:

1. ***Dom0 Host Baseline*** – Without any protections, the web-service runs on Dom0. This configuration is the baseline comparison measurement.
2. ***Virtual Guest Baseline*** – Without any protections, the web-service runs inside a guest VM. This configuration includes Xen VM costs.
3. ***Storage Crumple Zone (SCZ)*** – With preventive SCZ protections, the web-service runs in a container. This configuration includes storage channel mediation and includes filtering costs.
4. ***Network Crumple Zone (NCZ)*** – Building upon the previous configuration, add a NCZ with a canary-replica to taste-test and proxy Apache inputs to the WS. This configuration incorporates network channel mediation and proxy cost.
5. ***Network Crumple Zone w/ Filters*** – Building upon the previous configuration, enable HTTP filtering for web-service requests. This configuration adds ingress filtering cost on the network channel.
6. ***Guest Virtual Machine Introspection*** – Building upon the previous configuration, enable VMI monitoring for the protected application.

Fig. 5 shows the request latencies (y-axis) for groups of concurrent clients and various configurations of the A3 environment (x-axis). Latency results are normalized by removing outliers greater than three standard-deviations from each experiment's mean. In the worst case experiment, this is less than a 2.8% reduction in the count of experimental observations, implying experimental jitter was minimal. At first glance, Fig. 5 shows a pattern of increased cost as the number of concurrent clients and preventive protections are increased. As expected, even the virtualization cost increases from the external client's perspective when evaluating the web-service in a virtual guest compared to on the host (BASELINE-FC8-VM versus BASELINE-FC8-Dom0). The virtualization cost can be attributed to the experiment load over fixed quality-of-service guarantees for the guest VM (i.e., 1VCPU, 512MB Memory). While the Xen-based prototype is acceptable for research and evaluation, this observation implies that light-weight containerization alternates such as OS virtualization would benefit A3.

At low loads (1 and 3 clients), the SCZ mediation introduces a modest cost of 44% and 28% over the BASELINE-FC8-VM, and increases to 160% and 183% in the five and seven client configurations. Storage costs are attributable to the SCZ's UNFS implementation, which is not used in either of the baseline experiments. UNFS offers A3 a strong degree of isolation and a nice interception point to mediate storage operations. As an alternative, we could use closer to application techniques such as system call interception to drive down storage mediation costs, but at the price of losing the strong isolation from external SCZs in A3.

The next configuration (SCZ+NCZ+PROXY) introduces NCZ mediation. The NCZ implements a full application proxy to intercept and execute HTTP requests. If the execution does not trigger a bad effect, the initial request will be proxied to the protected application. Use of the full-application proxy incurs over twice the cost in processing an application request, as all requests are processed in both the application and NCZ domains. Furthermore, since the NCZ and application domains are backed by the SCZ, the storage channel traffic is effectively doubled. In this configuration, we see a 7.7%, 32.8%, 39.7% and 104.8% increase in costs as clients are scaled. When further enabling the NCZ filters (SCZ+NCZ+PROXY+FILTER), deep inspection of the HTTP payload increases the cost 19.9%, 30.8%, 17.5% and 15.6% respectively over the non-filtered case. This

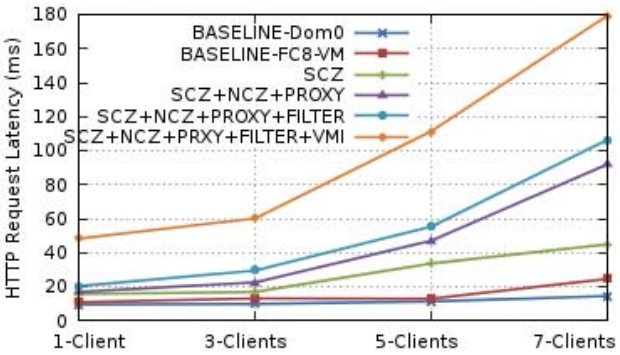

Fig. 5 – Supporting Service Overheads

observation leads us to believe that that some of the network filtering costs will be absorbed by the machinery implementing the proxying functionality. While the efficiency of the full-application proxy is poor, the value of the NCZ's full-application proxy lies in its ability to absorb the attack before it touches the real application. The proxy will further inform A3's adaptive procedures to help diagnose underlying problems, starting with Input-Reduction.

In the final experiment, VMI probes are used to monitor a select set of system calls against the protected application. VMI is by far the greatest contributor to preventive protection cost, but is also the most informative to Input-Reduction. As client count scales, VMI increases client latencies 137%, 103%, 101% and 68% over the network filtering experiment. It is interesting to note that the latency percentage decreases at each step as clients are added. Upon inspection, this behavior is explained through the blocking read semantics of the fixed-size client networking pool that is used to test the web-service. In practice, blocking reads will regulate the specified client request frequencies, effectively limiting the number of requests sent to the server and load. Upon further examination of client invocation data collected (not graphed), we do in-fact see a 91%, 83%, 85% and 76% percent reduction in total number of HTTP GET requests between the two configurations.

The poor performance of VMI is attributed to its use of software break pointing, like most debuggers. VMI-based preventive protection in A3 is more costly than the protections offered by crumple zones. On the other hand, VMI protections are independent of the untrusted application containers. Thus, VMI can provide a stronger security guarantee, and watch the application and inform Input-Reduction in ways that crumple zones cannot. In the next section we further decompose and assess VMI costs.

C. Composable Costs of Input-Reduction

Detection provided by VMI monitoring, is instrumental for deep diagnostic inspection of a protected application and further informing Input-Reduction. However as shown in previous sections experimentation, VMI is also the greatest contributor to preventive protection costs. In this sub-section, we break down VMI costs. We further consider the cost implication of deep inspection, with an eye towards improving the vmprobesys design for Input-Reduction.

At runtime, we have used VMI to monitor *execve*, *waitpid*, *fork*, *clone*, *socketcall* and *exit* entry points into the kernel. We further filter *accept*, *bind*, *connect*, *getpeername* and *getsockname* calls invoked through the multiplexed sys_socketcall interface on 32-bit Linux. Monitoring the system call interface, VMI will capture all system call operations whether they originate from the protected application, or from some other application executing in the guest VM. This implies that the VMI cost in the previous section is not only attributed to load on the protected web-service, but also on the whole host. To quantify this observation, we profile the system calls made by the web-service using the strace tool on Linux, and compare the call counts with the number of intercepted VMI calls in Table 2.

The non-starred section in Table 2 lists the six probed system calls, prefixed by sys_, while the shaded section lists six filtered sub-groupings of sys_socketcall invocations. The first five starred subgroups reflect the exact filtering specification in Fig. 2, while the *other socket calls* entry (last row) counts any other unrelated socketcall invocations which are trapped. The counts indicate that during a three minute experiment with 7 clients, vmprobesys intercepts extraneous system calls that do not originate from the protected application, but undoubtedly impact performance. The dominant interception point is the sys_socketcall interface registering 30,280 interceptions, of which 45% of calls are not attributed to the Apache process tree (i.e., Apache+PHP).

Cost attributed to VMI is dependent on three factors: the number of probes, the complexity of the filtering logic in each probe and the applications use of the probe point. Given this, we examine the effects of probes and filtering logic on client request latencies in two experiments, and then describe improvements that can be made to vmprobesys in future work. In our first experiment, in Fig. 6, we examine the effects of probe counts across four configurations, where we gradually scale back the number of system call probes. The *full configuration* includes the six system call probes in Table 2. The second configuration (labeled *five*) we remove the sys_socketcall probe and filters, configuration *four* removes the sys_clone call, and configuration *one* leaves only the sys_execve call. We also ran configurations with three and two probes, but they add little value to these results and are not shown. Once again, the reduced number of invocations in the full configuration is attributed to the blocking read client semantics when invoking the web-service application (described in IV.B).

Fig. 6 shows that the sys_socketcall probe is the dominant factor for the web-service application. Its removal reduces the average client latencies by 43%, and leads to a greater degree of stability in the variance of client latencies (labeled 'std'), whereas removal of the other probes lead to little positive change. This observation is attributed to the lower frequency of interceptions for those probes, and suggests that cost will mostly be a function of the *number of traversed probes*. Probe access is dictated by the nature of the protected application, here, a web-service and its use of sys_socketcall. We also conclude that non-dominant probes

Table 2. Application and VMI System Calls

32-bit Syscall (*socketcall subset)	strace –f of Apache	VMI Syscall Probes
sys_execve	8	12
sys_waitpid	430	503
sys_exit	0	13
sys_fork	0	0
sys_clone	17	98
sys_socketcall	N/A	30280
*accept	6697	N/A
*bind	8	N/A
*connect	3393	N/A
*getpeername	7	N/A
*getsockname	3382	N/A
*other socket calls	16793	N/A

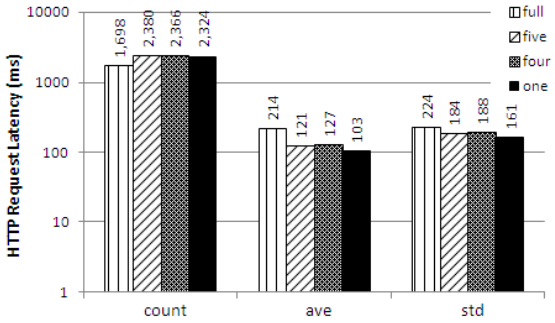

Fig. 6 - VMI Probe Effects

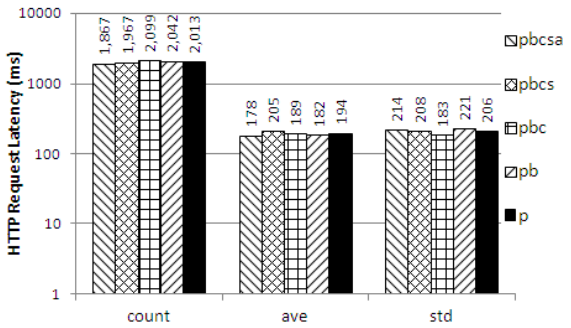

Fig. 7 - VMI Filter Effects

will add little incremental cost over a single dominant probe.

In the second VMI experiment, we varied the filtering logic using only sys_socketcall probes to understand the cost impact of filtering complexity. Filtering complexity includes both the cost associated with content parsing and pattern matching on the interface, and the local network cost of reporting to the A3 controller. Starting with five filters, we sequentially remove the accept filter first, and then the getsockname, connect, and bind filters, leaving only the getpeername filter in the final configuration. Fig. 7 shows that the filtering and reporting cost has no considerable effect on the client's request-response latency. More to the point, the probes existence is the dominant cost driver, not the conditional selection of the filters or upstream reporting. We conclude that complex filtering of even the most traversed probes do not add too much incremental cost, once the probes existence is required for defense.

To date, we have not attempted to optimize the VMI implementation or probe selection. We leave this to future work. To varying degrees, the effects of extraneous interceptions may be mitigated by inserting probes on demultiplexed kernel functions residing behind the multiplexed socket interface, or move to 64-bit Linux, were socket calls are not multiplexed. This would further reduce cost, but will not address the fact that the system call interface is a shared resource in the host container. A third option, which we are actively pursuing, is to develop a user-space probe target for vmprobesys. A user-space target will allow vmprobesys to breakpoint the application logic that triggers system call invocations, making VMI monitoring even more powerful. We believe this will improve both the accuracy and performance of A3 and Input-Reduction.

V. Related Work

Input-Reduction is a form of delta debugging [13, 15, 18], an automatic software testing approach that aims to find a minimal application input that reproduces a previously observed failure. Starting from a failure-causing input, a delta debugger generates and tests input variations that have one or more input subsequences removed; the search ends when no further input elements can be removed while maintaining the observed failure. Delta is a general search technique, and as such it makes no special considerations for minimizing state resets for systems under test, or any assumptions about the location of failure-relevant inputs. Input-Reduction, on the other hand, attempts to quickly isolate discrete message inputs from an input sequence under the assumption that state resets are expensive and that failure-causing inputs are likely to be near the end of a recorded trace (i.e., temporally near an observed fault).

Previous research systems have incorporated delta debugging into software execution and analysis environments. For example, like A3, the Malfor system [3] uses record-and-replay to capture inputs and detect security-relevant behavior of a monitored application. Whereas A3's Input-Reduction algorithm seeks to minimize the security-relevant input, Malfor seeks to identify the security-relevant processes within the monitored system; Neuhaus and Zeller stated that input reduction was planned as a future Malfor extension. Another environment that uses delta debugging is Triage [16], an environment that couples delta, replay, and common debugging tools for the purpose of performing software fault diagnosis at production sites. Neither the Malfor nor the Triage systems aim to support subsequent adaptive procedures informed by the results of input reduction for purposes of survivable execution.

As a whole, adaptive execution environments incorporating per-application containerization, application-focused monitoring, and adaptive defenses based upon replay are generally not delivered as turn-key solutions. Existing research addresses individual portions of A3's goals, but is often developed in isolation and for specific purposes. This leaves users to integrate disparate components when promoting survivability. Research and development to transparently provide security measures via mediation of application I/O [14, 20], closely monitor access to CPU resources by splitting applications into isolated compartments [4], and isolate individual applications into per-VM containers [11] all fulfill individual parts of A3's containerization and mediation goals. While there has been significant research into replay techniques [5, 7, 8, 10, 17] and its use for identifying defects through techniques such as omniscient and replay debugging [23, 24], few take an end-to-end view of *replay experimentation* as it applies to continually improving the resiliency of a contained application. A3's ongoing work is informed by the wealth of information these works have uncovered concerning the complexities of replay.

VI. Conclusion and Future Work

In this paper, we describe Input-Reduction as a technique for isolating attack inputs from other innocuous, recorded inputs. A concrete realization of Input-Reduction in the A3 execution management environment is presented and evaluated using attacks against a representative web-service application. We further present a comprehensive evaluation of the cost associated with the infrastructure services that are necessary to perform Input-Reduction in A3.

Experiments demonstrate that Input-Reduction can be quite effective and efficient at removing innocuous inputs from a recording. In four experiments, Input-Reduction is capable of finding and isolating both non-dependent and dependent attack inputs within a reasonable complexity and time bound. Many attributes of Input-Reduction are tunable, and knobs such as end-of-cycle timeouts and inter-input replay speeds will have the largest effect on accuracy of Input-Reduction. To enhance both the efficiency and accuracy of Input-Reduction, we anticipate running Input-Reduction twice, once with short end-of-cycle timeouts and fast replay, and a second time where replay speed and timeouts are widened to promote better accuracy.

We find that VMI monitoring and analyses are the dominant factor affecting application scalability and performance in A3. We further demonstrate that VMI probes may be used selectively at run-time to minimize costs to client request latencies, albeit, selective coverage decreases monitoring of the protected application. We further describe how the use of VMI analyses may also be relegated to deep diagnostic inspection during replay experiments, where a premium is placed on reconstructing details of a successful attack in-order to quickly derive a successful patch.

In future work, we plan on extending the breadth and depth of A3's replay experiment portfolio, for the purposes of automating attack diagnostics and patch finding. For Input-Reduction, this goal implies revisiting the auto-tuning capabilities to enhance the accuracy of the technique. For diagnostic experiments, we are currently developing an extensible analysis infrastructure and a suite of configurable VMI-based analyses that may be used to finger-print the nature of an arbitrary attack. For variant experiments, we are developing an application build-and-test service and extensible crumple zone filter plug-in interface to automatically test changes to an application and its preventive policies.

VII. References

[1] P. Pal, R. Schantz, A. Paulos, J. Regehr, and M. Hibler. "Advanced Adaptive Application (A3) Environment: initial experience," Proc. of the Middleware. Industry Track Workshop. ACM, 2011. 8 p.

[2] P., Pal, et al. "A3: An Environment for Self-Adaptive Diagnosis and Immunization of Novel Attacks," Sixth Intl. Conf. on Self-Adaptive and Self-Organizing Systems Workshops. IEEE, 2012. pp. 15-22.

[3] S. Neuhaus and A. Zeller. "Isolating Intrusions by Automatic Experiments," Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS), Feb. 2006.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. "Wedge: splitting applications into reduced-privilege compartments". Proc. of the 5th USENIX Symp. on Networked Systems Design and Implementation (NSDI). USENIX Association, 2008. pp. 309-322.

[5] G. Dunlap, S. King, S. Cinar, M. Basrai, and Peter M. Chen. "Revirt: enabling intrusion analysis through virtual-machine logging and replay," Proc. of the 5th symposium on Operating Systems Design and Implementation (OSDI). ACM, 2002. pp. 211-224.

[6] CVE-2012-0021. Web. Feb 2013 <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0021>

[7] S. King and P. Chen. "Backtracking intrusions," Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP). ACM, 2003. pp. 223-236.

[8] D. Lee, et al. "Respec: efficient online multiprocessor replay via speculation and external determinism," Proc. of the 15th edition of Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2010. pp. 77-90.

[9] Libpcap. Web. Feb 2013. <www.sourceforge.net/projects/libpcap/>

[10] J. Newsome, D. Brumley, J. Franklin, and D. Song. "Replayer: automatic protocol replay by binary analysis," Proc. of the 13th ACM conference on Computer and Communications Security (CCS). ACM, 2006. pp. 311-321.

[11] J. Rutkowska. R Wojtczuk. Qubes OS Architecture. Jan. 2010. <qubes-os.org/files/doc/arch-spec-0.3.pdf>

[12] CWE/SANS TOP 25 Most Dangerous Software Errors. Web. 27 June 2011. <www.sans.org/top25-software-errors/>

[13] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 28(2):183–200, February 2002.

[14] M. Sherr and M. Blaze. "Application containers without virtual machines," Proceedings of the 1st ACM workshop on Virtual machine security. ACM, 2009. pp. 39-42.

[15] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In Proc. of the 28th Intl. Conf. on Software Engineering (ICSE), Shanghai, China, May 2006. pp. 142–151.

[16] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. "Triage: diagnosing production run failures at the user's site," In Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP). ACM 2007. pp. 131-144.

[17] K. Veeraraghavan, et al. "Doubleplay: parallelizing sequential logging and replay," In Proc. of the 16th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2011. pp. 15-26.

[18] S. McPeak and D. Wilkerson. Delta. Web. 2003. http://delta.tigris.org/.

[19] XenAccess – A virtual machine introspection library for Xen. Web. Feb 2013. <http://code.google.com/p/xenaccess/>

[20] SquidGuard. Web. Feb 2013. <http://www.squidguard.org>

[21] P. Barham, et al. 2003. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (October 2003), 164-177.

[22] Syrian Web. Feb 2013. http://packetstormsecurity.com/files/view/102849/syrian-shell.tgz

[23] Y. Khoo, J. Foster, and M. Hicks. "Expositor: scriptable time-travel debugging with first-class traces." In Proc. of the 2013 Intl. Conf. on Software Engineering (ICSE '13). IEEE Press, 2013. pp. 352-361.

[24] A. Visan, K. Arya, G. Cooperman, and T. Denniston. "URDB: a universal reversible debugger based on decomposing debugging histories." In Proc. of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11).