FUZZBUSTER: A System for Self-Adaptive Immunity from Cyber Threats

David J. Musliner, Jeffrey M. Rye, Dan Thomsen, David D. McDonald, Mark H. Burstein
Smart Information Flow Technologies (SIFT)
Minneapolis, MN, USA
Email: {musliner, rye, dthomsen, dmcdonald, burstein}@sift.net

Paul Robertson
Dynamic Object Language Labs
Boston, MA, USA
Email: paulr@dollabs.com

***Abstract*—Today's computer systems are under relentless attack from cyber attackers armed with sophisticated vulnerability search and exploit development toolkits. To protect against such threats, we are developing FUZZBUSTER, an automated system that provides adaptive immunity against a wide variety of cyber threats. FUZZBUSTER reacts to observed attacks and proactively searches for never-before-seen vulnerabilities. FUZZBUSTER uses a suite of fuzz testing and vulnerability assessment tools to find or verify the existence of vulnerabilities. Then FUZZBUSTER conducts additional tests to characterize the extent of the vulnerability, identifying ways it can be triggered. After characterizing a vulnerability, FUZZBUSTER synthesizes and applies an adaptation to prevent future exploits.**

***Keywords*-self-adaptive immunity, cyber-security, fuzz-testing.**

I. INTRODUCTION

Modern computer systems face constant attack from sophisticated adversaries, and the number of cyber-intrusions increases every year [1], [2]. Cyber-attackers use numerous vulnerability scanning tools that automatically probe target software systems for a wide array of vulnerabilities. For example, attackers use fuzz-testing tools (such as Peach and SPIKE) that try to crash target applications, and SQL injection tools (such as sqlmap and havij) that attempt to manipulate the contents of databases. Upon discovering a potential vulnerability, attackers use powerful exploit development toolkits (such as Metasploit and Inguma) to quickly craft exploits that take advantage of identified vulnerabilities.

Under DARPA's Clean-slate design of Resilient, Adaptive, Survivable Hosts (CRASH) program, we are developing FUZZBUSTER to provide adaptive immunity from these and other cyber-threats. FUZZBUSTER provides long-term immunity against both observed and novel (zero-day) cyber-attacks.

As shown in Figure 1, FUZZBUSTER operates *proactively* to find vulnerabilities before they can be exploited, and *reactively* to address exploits observed "in the wild." FUZZBUSTER directs the execution of custom and off-the-shelf *fuzz-testing* tools to find and characterize vulnerabilities. Fuzz-testing tools find software vulnerabilities by exploring millions of semi-random inputs to a program. Given time and expert guidance, fuzz-testing has proven effective at finding a wide variety of software flaws, including defects that account for the most severe security problems [3].

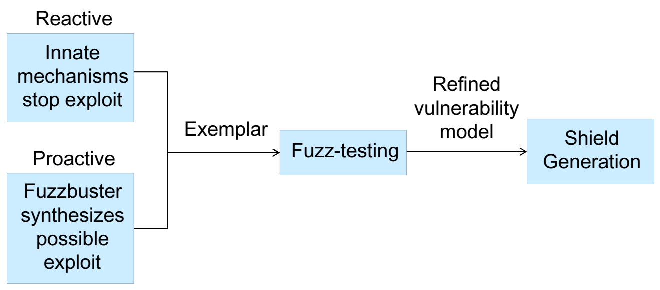
Reactive
Innate mechanisms stop exploit
Proactive
Fuzzbuster synthesizes possible exploit
Exemplar
Fuzz-testing
Refined vulnerability model
Shield Generation

Figure 1. When reacting to a fault, FUZZBUSTER creates an exemplar test case that reflects the environment and inputs at the time of the observed fault. During proactive exploration, FUZZBUSTER synthesizes exemplar test cases that could lead to a fault.

FUZZBUSTER uses fuzz-testing tools to find and characterize vulnerabilities, determining what inputs to a program can cause a fault. FUZZBUSTER then synthesizes defenses to shield or repair the flaw, protecting against entire classes of exploits that may be encountered in the future.

In this paper, we describe our rapidly-evolving implementation of the FUZZBUSTER architecture, and present some preliminary results.

II. ARCHITECTURE

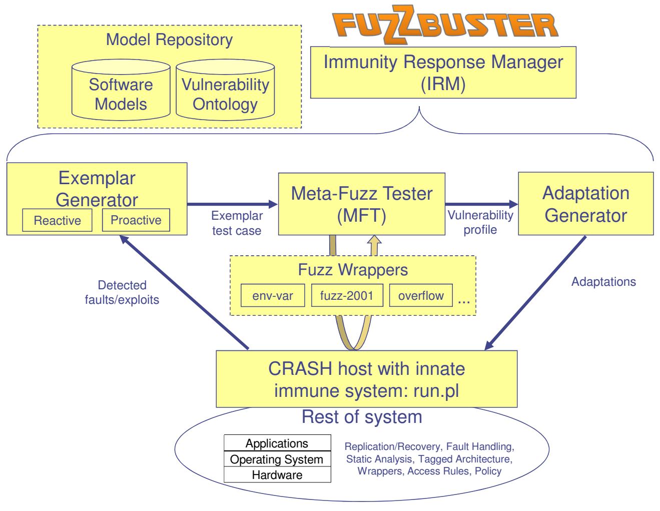

Figure 2. FUZZBUSTER's IRM guides its efforts to automatically find, refine, and adaptively shield vulnerabilities.

Figure 2 illustrates FUZZBUSTER's major components and how they interact to provide adaptive immunity. FUZZBUSTER uses both proactive and reactive exploration to identify (and then shield) vulnerabilities in a CRASH host. For each vulnerability, FUZZBUSTER creates a *vulnerability profile* representing the nature of the vulnerability, including what ranges of inputs lead to the vulnerability. These vulnerability profiles represent as much of the vulnerability as FUZZBUSTER can identify. After constructing a vulnerability profile, FUZZBUSTER creates and applies an adaptation that prevents future exploits of the vulnerability.

Processing begins with the Exemplar Generator creating an exemplar test case. The Exemplar Generator may create an exemplar in response to a fault notification from the CRASH innate immune system or in response to an instruction from the Immunity Response Manager (IRM) to initiate proactive exploration. At some point, the IRM determines that looking for vulnerabilities relating to a particular exemplar test case is the next highest priority activity, and the IRM assigns this activity to the Meta-Fuzz Tester (MFT). Based on the nature and attributes of the exemplar test case, the MFT chooses a fuzz-testing tool to search for or assess vulnerabilities associated with the exemplar test case. Each fuzz-testing tool refines a vulnerability profile based on the results of its exploration. The MFT may use multiple fuzz-testing tools to construct as complete a vulnerability profile as possible given the available time or resources. For each vulnerability profile, the Adaptation Generator creates one or more candidate adaptations to protect the system against being exploited via the vulnerability. When appropriate, the IRM directs the Adaptation Generator to verify and then subsequently apply these patches to the CRASH host. Since fuzz-testing and patch verification both run tests that may require significant time or resources to complete, the IRM is intended to balance the priorities of these operations with the available resources on the system, to minimize FUZZBUSTER's impact on system performance.

When an actual exploit or flaw is encountered and trapped by the CRASH innate immune system, FUZZBUSTER responds reactively. In our design, the IRM puts a high priority on responding to a live exploit, and may immediately choose to use the Adaptation Generator to synthesize a customized adaptation to shield the application while also engaging the MFT to refine the vulnerability profile. FUZZBUSTER may be conservative when reacting to an exploit, initially disabling useful features of the subject software while disabling the vulnerability. As the tests yield additional information, FUZZBUSTER revises the adaptation to relax (or tighten) the behavior restrictions it enforces. In this way, FUZZBUSTER acts as a self-protecting, self-regenerative system, initially clamping down on security and limiting functions when attacked, and then gradually relaxing limits and restoring functions as it gets a better picture of the vulnerabilities that are being exploited.

A. *Infrastructure*

1) GBBopen: Our FUZZBUSTER implementation is built within the GBBopen blackboard system [4], [5], which supports object-oriented data storage and event-triggered procedural code. The functional components shown in Figure 2 are implemented as blackboard Knowledge Sources (KSs) that respond to events on blackboard objects representing the ongoing tasks and results.

For instance, exemplar objects are used to describe cases raised by the immune system and cases where proactive exploration is suggested. Vulnerability profile objects capture a progressively-refined model of the set of situations that leading to security warnings for a particular software system.

During FUZZBUSTER's exploration, it often needs to initiate resource-intensive testing tasks or perform operations that change the behavior of the CRASH host. To prevent these activities from executing concurrently, and thus interfering with each other, FUZZBUSTER defines a set of task objects that are managed by the IRM. The processing of these tasks is started and stopped by the IRM as necessary to ensure effective operation of the system and to prevent conflicts between concurrent activities. The KSs that perform the processing have specific code to manage task status and their own work. In the near future, we will replace this *ad hoc* mechanism with a more automated and rigorous meta-control scheme. The more complete meta-control scheme will control the starting and stopping of tasks to ensure consistent and correct behavior, while easing the effort required to specify the desired properties and interactions between them.

2) Interface to CRASH Host (`run.pl`): FUZZBUSTER is designed to run in the context of a CRASH host whose innate immune system provides alerts to violations that may indicate vulnerabilities. Since a physical instantiation of the CRASH host is not yet available, FUZZBUSTER defines a proxy that serves as a stand-in. The proxy is currently implemented as a Perl script that provides key CRASH functionality on existing systems. In particular, the proxy mimics the CRASH innate immune system by identifying and reporting certain classes of faults. The proxy also provides an adaptation mechanism that allows FUZZBUSTER to modify the environment and inputs of executing programs.

B. *Exemplar Generator*

The Exemplar Generator captures relevant inputs and environmental aspects of an observed or suspected vulnerability as an exemplar test case. Ideally, an exemplar test case contains all of the information required to generate a repeatable test case for the MFT. However, since it is not always possible to record every relevant piece of information, and knowing the relevant bits is impossible in the context of proactive exploration, the MFT treats an exemplar test case as a starting point for exploration. An exemplar test case may also indicate that it pertains to a frequently-observed attack or applies to a mission-critical software system, which the IRM will use to set a suitably high priority on the discovery of the vulnerability and the implementation of a defense.

When the CRASH proxy `run.pl` detects a fault, it sends the Exemplar Generator a description of the environment and inputs that triggered the fault. The description includes each environment variable and its value, the path of the command, the command line arguments, and the content passed thru open streams including standard-input.

For proactive exploration, the Exemplar Generator also synthesizes exemplar test cases from application models. FUZZBUSTER stores models of applications that include the absolute path to the command, a specification of the allowed (or expected) command line arguments, and a flag indicating whether the application accepts input via standard-input. The Exemplar Generator translates these application models into exemplar test cases by choosing specific command line arguments or inputs.

C. Meta-Fuzz Tester

The Immunity Response Manager invokes the MFT to conduct an analysis of subsystem or protocol vulnerabilities, focused by the exemplar test case and limited by some computing resource constraints (initially, just execution time). The MFT attempts to identify the specific cause of an observed defect, or probe for a latent vulnerability in the case of proactive analysis. Starting from the exemplar test case, the MFT constructs a belief state describing the vulnerabilities that could be present. Then, as long as the MFT has remaining resources, it chooses a fuzz-testing tool and uses it to try to gain more information about the possible vulnerability. This analysis culminates in a vulnerability profile describing the observed aspects of the vulnerability and providing a basis for the Adaptation Generator to generate an adaptation that protects the system. The choice of fuzz-testing tool should be guided by a "performance profile" model of each tool's capabilities, in terms of what types of vulnerabilities they can detect and how long they may take. Our early experiments, discussed below, will help develop those performance profiles. In the meantime, our preliminary implementation uses a simpler method to allocate effort to different fuzz-testing tools.

To facilitate the integration of diverse fuzz-testing tools, FUZZBUSTER defines a *fuzz-tool wrapper* interface to each tool, providing a common API for controlling tool execution. Each fuzz-tool wrapper defines an action that may be taken by the MFT. Moreover, each wrapper interprets the results of execution, updating the vulnerability profile with additional information. Fuzz-tool wrappers provide a simple mechanism for FUZZBUSTER to incorporate dumb or smart tools, with or without knowledge of the internals of the test object.

D. Adaptation Generator

FUZZBUSTER's Adaptation Generator improves system security by creating and applying custom adaptations that prevent exploitation of the flaws characterized by vulnerability profiles. The Adaptation Generator uses a variety of adaptive techniques, making the choice between them based on an adaptation's needs and the facilities available for the relevant input channels. Our design anticipates that adaptations could be defined at any level in the system, from an atomic instruction, to a function call, to a high-level function of an application. Our initial implementation operates only at the application-input level, using the facilities provided by the `run.pl` CRASH proxy.

To safely adapt a live system, the Adaptation Generator follows two core principles. First, adaptations only restrict or reduce capability or privilege. Second, adaptations do not disable key functionality. To enforce the second principle, FUZZBUSTER will capture a set of test cases during vulnerability analysis. Some of these tests will trigger the vulnerability and others will exercise the vulnerable application without triggering the vulnerability. These tests will be used during adaptation creation to verify that an adaptation successfully prevents the vulnerability without otherwise changing the results or behavior of the vulnerable application. Once an adaptation is applied to the CRASH system, the tests will be added to a regression suite that FUZZBUSTER will use to ensure that future patches do not conflict with or invalidate existing adaptations.

The Adaptation Generator performs the following actions on adaptations:

- **Create —** Search for adaptations that make the best trade-off between performance, functional impact, and security.
- **Verify —** Execute recorded test cases to ensure that an adaptation prevents exploitation of a vulnerability without otherwise affecting execution.
- **Apply —** Apply adaptations to the system to prevent exploitation of vulnerabilities.
- **Revoke —** Remove previously applied adaptations from the system because they are no longer desirable, due to external software updates, more comprehensive adaptations, or to improve performance.

When creating an adaptation, the Adaptation Generator maps the constraints in the vulnerability profile to a set of actions that the adaptation can take to prevent the fault. FUZZBUSTER's initial set of actions includes "remove," "modify," "truncate," and "filter". An adaptation using the remove action completely removes the fault-inducing input, for instance unsetting an environmental variable. An adaptation using the modify action performs an arbitrary modification of the input, for example replacing the value with another one. The truncate and filter actions apply common modifications to inputs. Truncate reduces the size of the input channel to a specific threshold, for example shortening the length of an argument to prevent a buffer overflow. The filter action replaces specific substrings in the input channel. The Adaptation Generator examines the vulnerability profile to derive parameters for these actions, such as the target length for truncation or the content to remove.

When instructed by the IRM, the Adaptation Generator verifies an adaptation by temporarily applying the adaptation to the system and running the accumulated test-cases. Once an adaptation passes verification, the IRM may instruct the Adaptation Generator to apply it to the system, thus preventing a vulnerability from being exploited. An adaptation fails verification if it changes the behavior for non-fault-inducing inputs or if it fails to prevent a fault.

E. Immunity Response Manager

The IRM oversees and manages FUZZBUSTER's adaptive immunity processes, ensuring that FUZZBUSTER's proactive and reactive protection functions are effective, while avoiding undue burden on the resources of the protected system.

The IRM's chief roles include initiating proactive vulnerability exploration, assigning test priorities, and tasking the MFT and Adaptation Generator. Across these activities, the IRM controls the system by creating, assigning, and pausing tasks. Each task specifies a unit of work to be performed by a component in the system. FUZZBUSTER defines tasks for exploring an exemplar test case, verifying a patch, applying a patch to the system, and revoking a patch. By controlling which tasks are active, the IRM controls the balance between proactive and reactive testing, decides when to allocate resources to verifying that patches are acceptable, and controls when FUZZBUSTER modifies the system.

Our initial implementation of the IRM uses a hand-coded, static prioritization scheme that ranks tasks based on their order of arrival. This initial implementation ensures that FUZZBUSTER eventually explores all exemplar test cases and attempts to apply adaptations for all identified vulnerability profiles. In the future, the IRM will evolve into an MDP-based meta-controller similar to the approach described in [6].

III. Experimental Results

With the first version of each FUZZBUSTER module now functional, we have conducted numerous small tests and one significant series of long experiments. In those experiments, we used FUZZBUSTER to proactively search for vulnerabilities in a set of 53 command-line utilities. We ran the exploration on a Debian VM and a laptop running OS X; both systems were fully patched at the time of the experiment. FUZZBUSTER used a wrapper around Barton Miller's fuzz-testing tool to generate random byte sequences to feed to the programs being tested. For each trial, we configured FUZZBUSTER to use a specific set of options to the fuzz-testing tool, varying:

- whether the inputs could contain non-printable characters or not,
- whether the inputs could contain null characters or not,
- the initial seed to use for randomization, and
- the length of the input.

Each trial ran for a fixed period of time (usually 20 seconds), or until FUZZBUSTER found a fault. We relied on our CRASH stand-in (`run.pl`) to identify faults. For the purposes of this experiment, we identified faults as program crashes (abnormal exits, such as from segmentation faults).

FUZZBUSTER ran 3,380 trials in just over 18 hours, encountering 49 faults. Fifteen of those faults were "duplicates" caused by the same input as another trial but with additional, unnecessary, content at the end. For example, we found a fault in `tcsh` with a 1,000 byte input created with both printable and non-printable characters, no nulls, and a seed of 1,002; that same fault was subsequently encountered using a 10,000 byte input created with the same parameters. Another eleven faults differed only in that one fault was caused by feeding an input string to standard-input and the other was caused by feeding the same string via a file argument. The remaining 23 faults correspond to unique crashes in five programs: `a2p`, `dc`, `indent`, `tcsh`, and `troff`. FUZZBUSTER considers these to be unique vulnerabilities, as the inputs have unique combinations of printable/non-printable characters, presence of null characters, and seeds. However, we recognize that it is probable that these inputs are triggering fewer than 23 software problems, perhaps as few as five (one per program). Even if several of these faults stem from a single vulnerability, the full FUZZBUSTER will identify and shield the common vulnerability, thus protecting the system against the original and related faults.

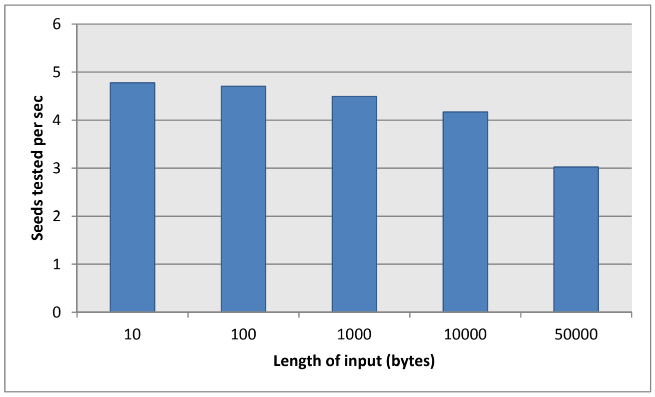

Figure 3. FUZZBUSTER executes 4.7 test cases per second when the inputs are short (10 to 100 bytes). The testing rate decreases with larger test inputs, falling to 3.0 tests per second when the inputs are 50,000 bytes long.

We configured FUZZBUSTER to repeat the proactive exploration numerous times, varying each of the conditions.

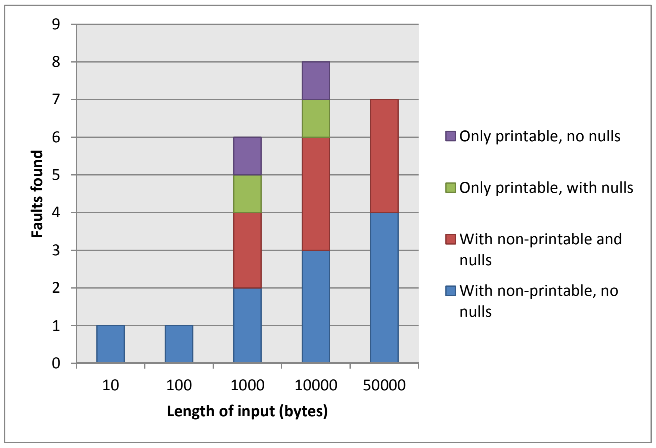

Figure 4. FUZZBUSTER requires larger input files to uncover all the faults encountered during the experiment. Inputs with non-printable characters led to discovery of 19 faults.

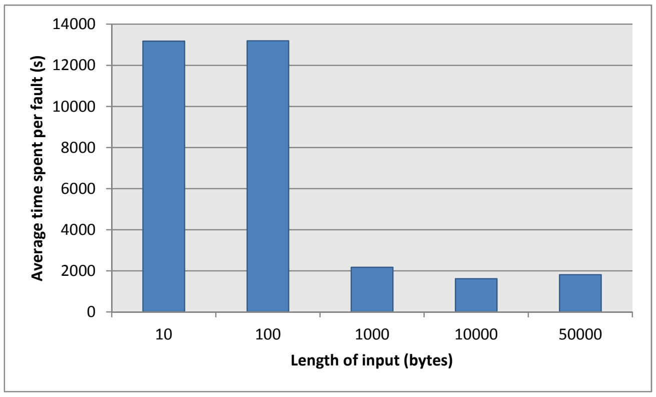

Figure 5. Even though short inputs result in faster test execution, due to the relative rarity of faults identified by short inputs, the average time spent searching per fault is much higher.

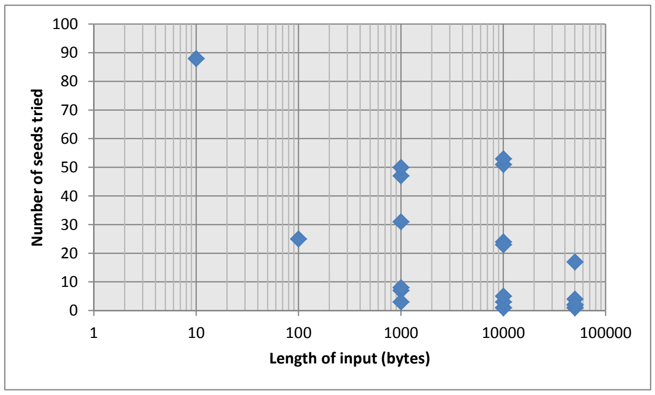

Figure 6. Number of seeds tested in a trial before finding one that induces a fault.

We tested with inputs of length 10, 100, 1,000, 10,000, and 50,000 bytes. As shown in Figure 3, running with the larger inputs modestly reduces the rate at which FUZZBUSTER runs individual test cases. Figure 4 shows that FUZZBUSTER usually needs inputs of at least 1,000 bytes to trigger the faults. Since we removed faults that were duplicates except for size, Figure 4 illustrates unique faults discovered at each size. Thus, FUZZBUSTER requires inputs with a length of 50,000 bytes to discover all of the faults in the test. From this graph we can also see that testing with non-printable characters is more effective than testing without, accounting for 19 out of the 23 faults (82.6%). Inputs containing null characters account of 10 faults and inputs without nulls account for 13.

While this suggests that FUZZBUSTER should focus on inputs containing non-printable characters, two applications (`a2p` and `indent`) *only* faulted when the inputs consisted entirely of printable characters. Moreover, our experiment encountered only a single fault in each of these applications. Thus, we can see the benefit of the MFT producing multiple actions for each fuzz-tool wrapper.

As shown in Figure 5, FUZZBUSTER discovers faults much more frequently using larger inputs. This graph illustrates how much more effective it is to test with larger inputs, despite the decrease in the number of test cases per second. We would like the MFT to try to optimize its use of limited fuzz-testing resources, so we're also interested in estimating how long FUZZBUSTER should run a test configuration before giving up and trying another fuzz-tool wrapper or abandoning the task. We can begin to answer this by examining how many inputs FUZZBUSTER tried before finding one that caused a fault. Figure 6 shows how many non-faulting seeds were tried in each trial that found a fault. The graph shows that it took, at most, 53 test-cases for inputs of 10,000 bytes and 50 test-cases for inputs of 1,000 bytes before finding the fault. Given the test rates from Figure 3, this shows that each of the successful tests took less than fifteen seconds, for larger inputs. While the precise test duration will vary according to the type of software being tested, these values provide a good starting point for our upcoming MDP-based Immunity Response Manager meta-controller.

IV. RELATED WORK

As previously noted, the FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 in the context of software security analysis [7]. It refers to invalid, random, or unexpected data that is deliberately provided as program input in order to identify defects. Fuzz-testers and the closely related "fault injectors" are good at finding buffer overflow, XSS, denial of service (DoS), SQL Injection, and format string bugs. They are generally not highly effective in finding vulnerabilities that do not cause program crashes, *e.g.*, encryption flaws and information disclosure vulnerabilities [8]. Moreover, existing fuzz-testing tools tend to rely significantly on expert user oversight, testing refinement, and decision-making in responding to identified vulnerabilities.

FUZZBUSTER is designed both to augment the power of fuzz-testing and to address some of its key limitations. FUZZBUSTER fully automates the process of identifying seeds for fuzz-testing, guides the use of fuzz-testing to develop general vulnerability profiles, and automates the synthesis of defenses for identified vulnerabilities.

To date, several research groups have created specialized self-adaptive systems for protecting software applications. For example, both AWDRAT [9] and PMOP [10] used dynamically-programmed wrappers to compare program activities against hand-generated models, detecting attacks and blocking them or adaptively selecting application methods to avoid damage or compromises.

The CORTEX system [11] used a different approach, placing a dynamically-programmed proxy in front of a replicated database server and using active experimentation based on learned (not hand-coded) models to diagnose new system vulnerabilities and protect against novel attacks.

While these systems demonstrated the feasibility of the self-adaptive, self-regenerative software concept, they are closely tailored to specific applications and specific representations of program behavior. FUZZBUSTER provides a general approach to adaptive immunity that is not limited to a single class of application. FUZZBUSTER does not require detailed system models, but will work from high-level descriptions of component interactions, such as APIs or contracts. Furthermore, FUZZBUSTER's proactive use of intelligent, automatic fuzz-testing identifies possible vulnerabilities before they can be exploited.

V. Conclusion and Future Work

FUZZBUSTER is intended to augment and eventually outmode various post-exploit security tools such as virus scanners. Rather than scanning a computer all night to see if it has been compromised by an exploit, FUZZBUSTER will scan for vulnerable software and repair or shield it. Our preliminary experiments have shown that there are still many such vulnerabilities to be found, even on heavily used software in mature systems. As we extend FUZZBUSTER to address more complex applications with more forms of input, we expect that FUZZBUSTER will find vulnerabilities even more frequently. We hope that FUZZBUSTER will play a crucial role in proactively finding and eliminating vulnerabilities, making fuzz-testing no longer an effective strategy for cyber-attackers.

Acknowledgments

This work was supported by DARPA and Air Force Research Laboratory under contract FA8650-10-C-7087. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] T. Kellerman, "Cyber-threat proliferation: Today's truly pervasive global epidemic," *Security Privacy, IEEE*, vol. 8, no. 3, pp. 70 –73, May-June 2010.

[2] G. C. Wilshusen, "Cyber threats and vulnerabilities place federal systems at risk: Testimony before the subcommittee on government management, organization and procurement," United States Government Accountability Office, Tech. Rep., May 2009.

[3] "Automated penetration testing with white-box fuzzing," 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic1

[4] D. D. Corkill, "Countdown to success: dynamic objects, gbb, and radarsat-1," *Commun. ACM*, vol. 40, pp. 48–58, May 1997.

[5] ——, "Blackboard systems," *AI expert*, vol. 6, no. 9, pp. 40–47, 1991.

[6] D. J. Musliner, R. P. Goldman, and K. D. Krebsbach, "Deliberation scheduling strategies for adaptive mission planning in real-time environments," in *Proc. Third International Workshop on Self Adaptive Software*, 2003.

[7] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, December 1990.

[8] C. Anley, J. Heasman, F. Linder, and G. Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Ed.* John Wiley & Sons, 2007, ch. The art of fuzzing.

[9] H. Shrobe, R. Laddaga, B. Balzer, N. Goldman, D. Wile, M. Tallis, T. Hollebeek, and A. Egyed, "AWDRAT: a cognitive middleware system for information survivability," *AI Magazine*, vol. 28, no. 3, p. 73, 2007.

[10] H. Shrobe, R. Laddaga, B. Balzer *et al.*, "Self-Adaptive systems for information survivability: PMOP and AWDRAT," in *Proc. First Int'l Conf. on Self-Adaptive and Self-Organizing Systems*, 2007, pp. 332–335.

[11] "Cortex: Mission-aware cognitive self-regeneration technology," Final Report, US Air Force Research Laboratories Contract Number FA8750-04-C-0253, March 2006.