Self-Adaptation Metrics for Active Cybersecurity

David J. Musliner, Scott E. Friedman, Tom Marble, Jeffrey M. Rye, Michael W. Boldt, Michael Pelican
Smart Information Flow Technologies (SIFT)
Minneapolis, MN, USA
Email: {dmusliner, sfriedman, tmarble, jrye, mboldt, mpelican}@sift.net

***Abstract*—FUZZBUSTER is a host-based adaptive security system that automatically discovers, refines, and repairs vulnerabilities in hosted applications in order to prevent cyberattacks. FUZZBUSTER must decide when to adapt its applications, when to revoke its previous adaptations, and when to sacrifice functionality to improve security. This requires an adaptation quality metric that captures (1) an application's susceptibility to cyberattacks and (2) an application's functionality, since adapting an application affects both of these factors. FUZZBUSTER uses different types of test cases to measure security and functionality. In this paper, we describe FUZZBUSTER's adaptation metrics and we present two different policies for balancing security and functionality. We provide empirical results comparing these policies, and we also demonstrate how FUZZBUSTER can temporarily sacrifice the functionality of hosted applications to increase host security, and then restore functionality when more favorable adaptations are found.**

Keywords-self-adaptive immunity, cybersecurity, fuzz-testing.

I. INTRODUCTION

Cyber-intrusions pose a constant threat to today's computer systems, and the number of intrusions increases every year [1], [2]. Cyber attackers use sophisticated tools to detect and exploit system vulnerabilities (*e.g.*, [3], [4]). This creates a demand for systems that can quickly react to observed exploits with automatic diagnosis and adaptation. Furthermore, if the system can proactively discover a vulnerability *before* an attacker exploits it, zero-day exploits might be entirely prevented.

We are developing FUZZBUSTER under DARPA's Clean-slate design of Resilient, Adaptive, Survivable Hosts (CRASH) program to provide self-adaptive immunity against cyber-attacks. For an in-depth discussion of FUZZBUSTER's capabilities, see [5], [6], [7]. FUZZBUSTER uses a diverse set of custom and off-the-shelf fuzz-testing tools to perform protective self-adaptation. Fuzz-testing tools find software vulnerabilities by exploring millions of semi-random inputs to a program. FUZZBUSTER also uses them to refine its understanding of known vulnerabilities, clarifying which types of inputs can trigger a vulnerability. FUZZBUSTER's behavior falls into two general classes, as illustrated in Figure 1:

1) *Proactive*: FUZZBUSTER discovers novel vulnerabilities in applications using fuzz-testing tools. It refines its models of the vulnerabilities and then repairs them or shields them before attackers find and exploit them.

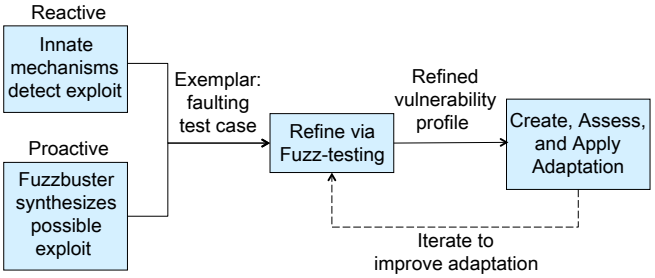

Figure 1. When reacting to an observed fault, FUZZBUSTER creates an exemplar that reflects the environment and inputs at the time of the fault. During proactive exploration, FUZZBUSTER synthesizes exemplars that could lead to a novel fault.

2) *Reactive*: FUZZBUSTER is notified of a fault in an application (potentially triggered by an adversary). FUZZBUSTER subsequently tries to refine the vulnerability and repair or shield it against attackers. Reactive vulnerabilities pose a greater threat to the host, since these may indicate an imminent exploit by an attacker.

FUZZBUSTER's primary objective is to protect its host by adapting its applications, but this may come at some cost. For example, applying an input filter or a binary patch may create a new vulnerability, re-enable a previously-addressed vulnerability, or otherwise negatively impact an application's usability by changing its expected behavior. This illustrates a tradeoff between functionality and security, and measuring both of these factors is important for making decisions about adaptive cybersecurity.

This paper describes how FUZZBUSTER evaluates the state of its applications with respect to security (*i.e.*, reactive and proactive vulnerabilities) and functionality. These evaluations determine when FUZZBUSTER applies a new adaptation or removes a prior adaptation.

We begin by outlining FUZZBUSTER's process of discovering, refining, and repairing vulnerabilities, which motivates our research on adaptation metrics. We then describe our approach and summarize the results of several experiments that demonstrate FUZZBUSTER's adaptation metrics in action.

A. FUZZBUSTER *Active Cybersecurity*

FUZZBUSTER tests and adapts multiple applications on its host. When FUZZBUSTER discovers a fault in one of

these applications— or when it is notified of a fault by its host— it represents the fault as an *exemplar* that contains information about the system's state when it faulted, as shown in Figure 1.

An exemplar includes information for replicating the fault, such as environment variables and data passed as input (*e.g.*, via sockets or `stdin`) to the faulting application. Some of this data may be unrelated to the underlying vulnerability (*e.g.*, the fault might be replicated without a specific environment variable binding). FUZZBUSTER uses the fuzz-testing tools in Table I to incrementally refine the exemplar, trying to characterize the minimal inputs needed to trigger the fault. Since time and processing power is limited, FUZZBUSTER uses a greedy meta-control strategy [7] to orchestrate these tools.

Refinement is an iterative process, where each task improves the *vulnerability profile* that FUZZBUSTER uses to characterize the vulnerability. The refinement process turns the initial (often over-specific) vulnerability profile into a more accurate and general profile.

FUZZBUSTER has several general adaptation capabilities, including input filters, environment variable filters, and source-code repair and recompilation. These protect against entire classes of exploits that may be encountered in the future. FUZZBUSTER uses each of these by (1) constructing the adaptation, (2) assessing the adaptation by temporarily applying it for test runs, and (3) applying the adaptation to the production application if it is deemed beneficial. FUZZBUSTER may apply multiple adaptations to an application to repair a single underlying vulnerability.

Note that FUZZBUSTER can create and apply adaptations *during* the vulnerability refinement process. So FUZZBUSTER can dynamically balance security against functionality, applying over-general adaptations initially to quickly shield a vulnerability, and then developing more specific and functionality-preserving adaptations as time permits [7].

In the next section, we describe FUZZBUSTER's metrics for evaluating new and existing adaptations and deciding when to apply, ignore, or remove them. We then present experimental evidence that these adaptation metrics help balance functionality with security.

Table I
FUZZ-TESTING TOOLS AND OTHER ACTIONS IN FUZZBUSTER.

Discovery actions replicate and discover vulnerabilities:

- `replicate-fault`: Given an exemplar from the host, replicate the fault under FUZZBUSTER's control.
- `gen-exemplar`: Generate an exemplar that might produce a fault.
- `fuzz-2001`: Generate random binary data and use it as input for `stdin`, file i/o, or command arguments.
- `cross-fuzz`: Use Javascript and the DOM to fuzz-test web browsers.
- `wfuzz`: Fuzz-test web servers with templated attacks.

Refinement actions improve vulnerability profiles:

- `env-var`: Identify environment variables that are necessary for a fault.
- `smallify`: Semi-randomly remove data from the faulting input to find faulting substring(s).
- `div-con`: Binary search for a smaller faulting input.
- `line-relev`: Remove unnecessary lines from multi-line faulting input.
- `find-regex`: Compute a regular expression to capture the faulting input.
- `insert-chars`: Insert characters to generalize regular expressions.
- `crest`: Given source code, use concolic search to find constraints on the faulting input [8].

Adaptation actions deploy a shield or repair a vulnerability:

- `create-patch`: Given a vulnerability profile, create a patch to filter input channels and environment variables.
- `verify-patch`: Assess a patch created by `create-patch` to ensure that it outperforms a security baseline.
- `apply-patch`: Apply a verified patch.
- `evolve-patch`: Given source code, use GenProg [9] to evolve a new non-faulting program source and binary.

II. ASSESSING ADAPTATIONS

FUZZBUSTER's adaptation metrics are based on *test cases*: mappings from application inputs (*e.g.*, sockets, `stdin`, command-line arguments, and environment variables) to application outputs (*e.g.*, `stdout` and return code). A *faulting test case* terminates with an error code or its execution time exceeds a set timeout parameter, while a *non-faulting test case* terminates gracefully. FUZZBUSTER stores three sets of test cases for each application under its control:

1) *Non-faulting test cases* (N). These test cases did not cause a fault when they were initially encountered. They include non-faulting test cases that FUZZBUSTER has created internally and non-faulting test cases that were supplied with an application for regression testing. N^+ is the subset of non-faulting test cases N with correct behavior (*i.e.*, output and return code), given some adaptations.
2) *Reactive faulting test cases* (R). These include reactive exemplars reported by FUZZBUSTER's host (see Figure 1) and other faulting test cases encountered while refining a reactive exemplar. Since these reflect a fault that FUZZBUSTER itself did not originally trigger, the underlying vulnerability may be known by adversaries. R^+ is the subset of reactive faulting test cases that are no longer faulting, given some adaptations.
3) *Proactive faulting test cases* (P). These include proactive exemplars discovered by FUZZBUSTER (see Figure 1) and other faulting test cases encountered while refining a proactive exemplar. The underlying vulnerability is less likely to be known by adversaries than in the reactive case. P^+ is the subset of proactive faulting test cases that are no longer faulting, given some adaptations.

Before FUZZBUSTER has discovered faults or been no-

tified of faults, there are no faulting test cases, so $R = P = R^+ = P^+ = \emptyset$ for all applications. As FUZZBUSTER proactively fuzzes its applications, many inputs will not fault, so the N sets will grow. As FUZZBUSTER encounters proactive and reactive faults, the R and P sets will grow, and as FUZZBUSTER subsequently refines these vulnerabilities (*e.g.*, by creating smaller faulting test cases), those sets will continue to grow.

FUZZBUSTER applies and removes adaptations to extend the sets R^+, P^+, and N^+. We have implemented two separate policies that guide FUZZBUSTER's adaptation behavior. We outline these policies and then describe experiments to evaluate their effectiveness.

A. Strict adaptation policy

Under the *strict* adaptation policy, when FUZZBUSTER creates an adaptation, it temporarily applies the adaptation and runs all of the non-faulting test cases N and the subset of faulting test cases (in P and R) that correspond to the current vulnerability profile. The adaptation is applied if it meets both of the following criteria:

1) All of the non-faulting test cases retain the same output and return code (*i.e.*, $N^+ = N$).
2) All of the faulting test cases are repaired.

Consequently, this policy only allows adaptations that are complete repairs and that preserve all known functionality, as determined by the test cases available. Once an adaptation is applied, it is never removed.

B. Relaxed adaptation policy

The *relaxed* policy allows FUZZBUSTER to sacrifice functionality to fix faulting test cases. Under the relaxed policy, fixing reactive faults (extending R^+) is highest priority, fixing proactive faults (extending P^+) is next, and maintaining the behavior of non-faulting test cases (extending or maintaining N^+) is lowest priority. FUZZBUSTER uses a dominant comparison function to compare application states, so it will apply or remove an adaptation to increase the size of R^+ regardless of the effect on P^+, and likewise for P^+ and N^+. So, when FUZZBUSTER compares two application states $s_1 = \langle R_1^+, P_1^+, N_1^+ \rangle$ and $s_2 = \langle R_2^+, P_2^+, N_2^+ \rangle$:

$$
\begin{aligned}
&if\ (R_1^+ > R_2^+)\ s_1 > s_2 \\
&if\ (R_1^+ = R_2^+) \wedge (P_1^+ > P_2^+)\ s_1 > s_2 \\
&if\ (R_1^+ = R_2^+) \wedge (P_1^+ = P_2^+) \wedge (N_1^+ > N_2^+)\ s_1\ > s_2 \\
&if\ (R_1^+ = R_2^+) \wedge (P_1^+ = P_2^+) \wedge (N_1^+ = N_2^+)\ s_1\ = s_2 \\
&else\ s_2\ > s_1
\end{aligned}
\tag{1}
$$

This allows FUZZBUSTER to compute strong (*e.g.*, $s_1 > s_2$) and equal (*e.g.*, $s_1 = s_2$) preferences between application states.

Before FUZZBUSTER applies an adaptation, it (1) computes the current application state s over test cases, (2) temporarily applies the adaptation and computes the resulting application state s', and (3) applies the adaptation if and only if $s' > s$. This means that each adaptation must provide some security or functionality benefit.

Furthermore, after a new adaptation is applied, FUZZBUSTER evaluates previously-applied adaptations for removal, by (1) computing the current application state s, (2) temporarily removing the adaptation in question and computing the resulting state s', and (3) removing it if and only if $s' \geq s$. This means each adaptation must *continally* provide security or functionality benefits, otherwise it will be removed.

We next describe experiments that compare these policies and illustrate the tradeoff of security and functionality.

III. EXPERIMENTS

We describe an empirical evaluation to compare strict and relaxed adaptation policies, and then we describe a scenario where FUZZBUSTER uses the relaxed policy to temporarily sacrifice functionality to increase security.

For all of these experiments, we provided FUZZBUSTER with a faulty version of `dc`, a Unix calculator program. This version of `dc` causes a segmentation fault when either (1) the modulo (%) operator is executed with at least two numbers on the stack or (2) base conversion is attempted with at least two numbers on the stack. Since many different input sequences will produce the fault, we do not expect a single adaptation to address the entire space of faults.

We also provided FUZZBUSTER 25 non-faulting test cases for `dc` to seed the N set, with the modulo and base conversion test cases removed, so that $N = N^+$. These were gathered from examples in the `dc` manual and the Wikipedia `dc` entry.

A. Comparing strict and relaxed policies

We ran FUZZBUSTER in strict mode and then in relaxed mode for 45 minutes using the experimental setup described above. In both cases, FUZZBUSTER uses its discovery tools to find vulnerabilities, its refinement tools to characterize the vulnerabilities, and its adaptation tools to create, assess, and apply adaptations.

1) Strict results: Figure 2 shows the results of FUZZBUSTER's strict policy. The solid red line plots $|P|$, the number of proactive faulting test cases found over time, due to discovery tasks and refinement tasks. The dashed red line plots $|P^+|$, the number of proactive test cases fixed over time due to adaptations. Both of these sets increase monotonically, and by definition, $|P|$ is the upper bound of $|P^+|$. The gap between $|P^+|$ and $|P|$ illustrates the share of test cases that FUZZBUSTER cannot repair under the strict policy over time. We call the area covered by this gap over

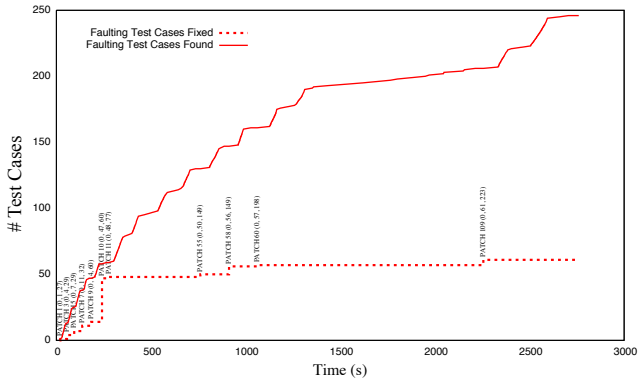

Figure 2. FUZZBUSTER uses the strict policy to preserve its applications' functionality throughout the course of protective adaptation.

time the *exposure* of the host to the vulnerability in question. Ideally, FUZZBUSTER will minimize this exposure.

By definition, the strict policy preserves all functionality of the application (*i.e.*, $|N^+| = |N|$), so we do not plot non-faulting test cases in Figure 2.

Adaptations are numbered sequentially, starting with "Patch 1" and increasing with each adaptation created. As shown in Figure 2, FUZZBUSTER created over 109 adaptations in 45 minutes, but only 11 passed the strict assessment criteria and were subsequently applied.

2) Relaxed results: Figure 3 shows the results of the relaxed policy. This is plotted identically to the strict results, except we also include the blue non-faulting dataset: the solid blue line plots $|N|$, the number of non-faulting test cases present; and the dotted blue line plots $|N^+|$, the number of non-faulting test cases with correct behavior. Non-faulting test cases are accrued over time by running discovery tasks that do not produce faults. The gap between $|N|$ and $|N^+|$ represents the *loss of functionality* over time. Like the exposure gap, we want to minimize the functionality gap.

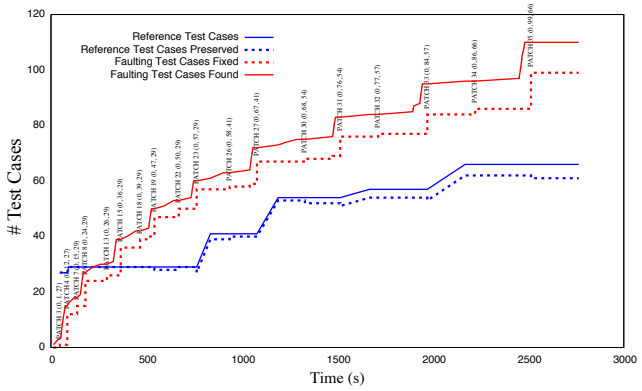

Figure 3. FUZZBUSTER uses the relaxed policy to sacrifice functionality for the sake of improving security.

As shown in Figure 3, FUZZBUSTER incurs a loss of functionality to remedy its exposure. More specifically, it sacrifices up to 10% of its non-faulting test cases to fix faulting test cases, and it restores the behavior of erronous non-faulting test-cases at multiple points, *e.g.*, around 600s and 950s.

There are several key differences between the strict and relaxed results:

- The P^+-to-P exposure gap is much smaller in relaxed mode than in strict mode, indicating more protection over time in relaxed mode.
- In relaxed mode, FUZZBUSTER applied 18 of 35 (51%) created adaptations, compared to 11 of 109 (10%) in strict mode. This means that relaxed mode wasted less time constructing unused adaptations.
- At the end of each run, relaxed mode yields $|P| = 110$ total faulting test cases, and strict mode yields $|P| = 245$ total faulting test cases. This is because FUZZBUSTER was unable to apply as many adaptations, so it discovered more variations of similar faulting test cases.

In relaxed mode, FUZZBUSTER does not fully restore `dc`'s functionality by the end of the 45 minute trial, nor does it restore the functionality after six hours – it retains a 10% loss of functionality. We discuss ideas for improvement in our conclusion.

B. Sacrificing and restoring functionality

We also ran FUZZBUSTER with a more agressive adaptation creation strategy, where its filters remove significantly more input. Figure 4 shows FUZZBUSTER's results after ten minutes, where overgeneral adaptations are applied to agressively close the exposure gap with considerable loss of functionality.

The graph is annotated to indicate where adaptations are applied to increase security, and where they are revoked to regain functionality.

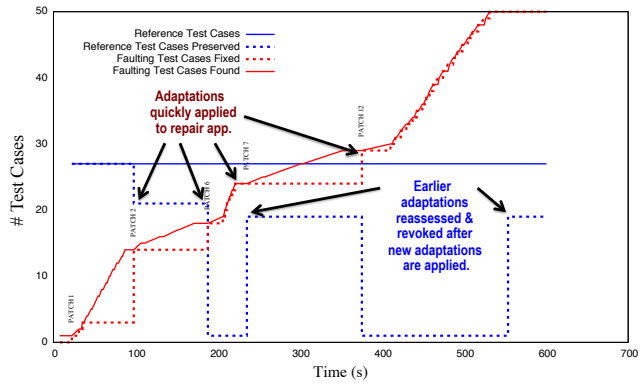

Figure 4. FUZZBUSTER uses the relaxed policy to apply overgeneral patches that temporarily sacrifice functionality. These patches are later revoked once more accurate adaptations are applied.

IV. Related Work

As previously noted, the FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 applied to software security analysis [10]. 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 [11]. 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 [12] and PMOP [13] 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 [14] 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 discover vulnerabilities and then quickly refine and adapt its applications to prevent them from being exploited by attackers. This requires representing the quality of adaptations to help determine when to apply, forego, and remove adaptations.

We described three different types of test cases FUZZBUSTER uses for quality assessment and two different adaptation policies that utilize these test cases: (1) the strict policy only applies adaptations that preserve functionality and repair a specific subset of faulting test cases, and (2) the relaxed policy applies adaptations to remedy (in decending priority) reactive faults, proactive faults, and application functionality, sacrificing lower-priority factors to address higher-priority factors.

We presented a comparison of these policies and demonstrated that the relaxed policy provides better protection over time and is more efficient in using the adaptations created by FUZZBUSTER. We also showed that, under the relaxed policy, FUZZBUSTER will temporarily sacrifice application functionality to increase host security: it aggressively applies over-general adaptations and then when more-specific adaptations are developed, it revokes the initial adaptations to restore some functionality.

Next steps for improving FUZZBUSTER's adaptation policies include improving its ability to fully restore functionality to its adapted applications. In our experiments, FUZZBUSTER did not fully restore the functionality of the adapted applications under the relaxed adaptation policy. We are extending and improving FUZZBUSTER's refinement tools to ultimately provide more accurate adaptations that have less impact on application functionality. We will also provide FUZZBUSTER with domain knowledge (*e.g.*, input channel specifications, character sets, and grammars) to make refinement more efficient and accurate. The metrics described above— and illustrated in the graphs of our experimental results— have the added benefit of helping us evaluate our FUZZBUSTER design decisions in the future.

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. Approved for public release, distribution unlimited.

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] "Metasploit framework penetration testing software." [Online]. Available: http://www.metasploit.com

[4] "Inguma." [Online]. Available: http://inguma-framework.org/projects/inguma

[5] D. J. Musliner, J. M. Rye, D. Thomsen, D. D. McDonald, and M. H. Burstein, "Fuzzbuster: Towards adaptive immunity from cyber threats," in *1st Awareness Workshop at the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, October 2011.

[6] D. J. Musliner, J. M. Rye, and T. Marble, "Using concolic testing to refine vulnerability profiles in fuzzbuster." in *SASO-12: Adaptive Host and Network Security Workshop at the Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, September 2012.

[7] D. J. Musliner, S. E. Friedman, J. M. Rye, and T. Marble, "Metacontrol for adaptive cybersecurity in fuzzbuster," in *SASO-13: The Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, September 2013.

[8] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.69

[9] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, May 2010.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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