Improving Automated Cybersecurity by Generalizing Faults and Quantifying Patch Performance

Scott E. Friedman, David J. Musliner, and Jeffrey M. Rye
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {sfriedman,dmusliner,jrye}@sift.net

***Abstract*—We are developing the FUZZBUSTER system to automatically identify software vulnerabilities and create adaptations that shield or repair those vulnerabilities before attackers can exploit them. FUZZBUSTER's goal is to minimize the time that vulnerabilities exist, while also preserving software functionality as much as possible. This paper presents new adaptive cybersecurity tools that we have integrated into FUZZBUSTER, as well as new metrics that FUZZBUSTER uses to assess their performance. FUZZBUSTER's new tools increase the efficiency of its diagnosis and adaptation operations, they produce more general, accurate adaptations, and they effectively preserve software functionality. We present FUZZBUSTER's analysis of 16 fault-injected command-line binaries and six previously known bugs in the Apache web server. We compare FUZZBUSTER's results for different adaptation strategies and tool settings, to characterize their benefits.**

***Keywords*-cyber defense, adaptive security, security metrics, filter generation.**

I. INTRODUCTION

Cyber-attackers constantly threaten today's computer systems, increasing the number of intrusions every year.Firewalls, anti-virus systems, and patch distribution systems react too slowly to newfound "zero-day" vulnerabilities, allowing intruders to wreak havoc. We are investigating ways to solve this problem by allowing computer systems to automatically identify their own vulnerabilities and adapt their software to shield or repair those vulnerabilities, before attackers can exploit them [1]. Such adaptations must balance the safety of the system against its functionality: the safest behavior might be to simply turn the power off or entirely disable vulnerable applications, but that would render the systems useless. To make a finer-grained balance between security and functionality, adaptations must be:

- General enough to shield the entire vulnerability (i.e., not just blocking an overspecific set of faulting inputs).
- Specific enough to minimize the negative impact on program functionality (e.g., by causing incorrect results on valid inputs).
- Efficiently-generated, to minimize the time during which a vulnerability is present or exposed.

These considerations for adaptive cybersecurity pose several challenges, including: how faults are discovered and diagnosed, with and without direct access to source code or binaries; how adaptations are generated from the diagnoses; how the many possible adaptations are assessed and chosen; and how all of these operations are orchestrated for efficiency.

This paper describes strategies for automatically discovering vulnerabilities, diagnosing them, and adapting programs to shield or repair those vulnerabilities. We have implemented these strategies within the FUZZBUSTER integrated system for adaptive cybersecurity [2], which includes metrics [3] and metacontrol [4] for self-adaptive software defense. FUZZBUSTER uses a diverse set of custom-built and off-the-shelf fuzz-testing tools and code analysis tools to develop protective self-adaptations. Fuzz-testing tools find software vulnerabilities by exploring millions of semi-random inputs to a program. FUZZBUSTER also uses fuzz-testing tools to refine its models 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. FUZZBUSTER refines its models of the vulnerabilities and then repairs them or shields them before attackers find and exploit them.
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.

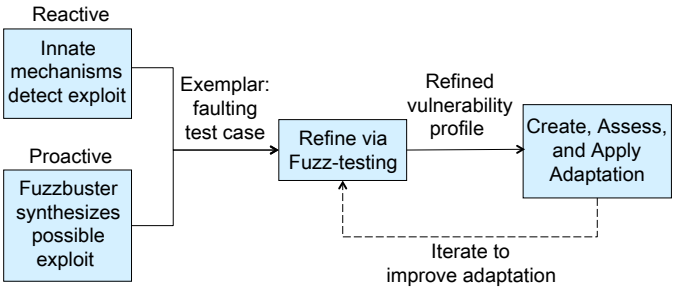

Fig. 1. FUZZBUSTER automatically finds vulnerabilities, refines its understanding of their extent, and creates adaptations to shield or repair them.

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.

We begin by outlining related work in Section II and describing FUZZBUSTER's process of discovering, refining, and repairing vulnerabilities in Section III. This supports the adaptation assessment metrics described in Section IV. We then describe FUZZBUSTER's novel diagnosis tools for adaptive cybersecurity in Section V and a novel fault-injection method for generating binaries for testing in Section VI. We evaluate FUZZBUSTER's performance on several experiments in Section VII on real and automatically-injected vulnerabilities in production-grade software. Section VIII summarizes our contributions and future research directions.

II. Related Work

The FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 applied to software security analysis [5]. 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, cross-site scripting, 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 [6]. 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 [7] and PMOP [8] 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 [9] 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.

Other adaptive cybersecurity work focuses on white-box program analysis instead of black-box fuzz-testing. Some of these defenses instrument binaries to change their execution semantics [10] or protect exploitable data [11]. Other approaches adapt binaries to add diversity at compile-time [12] or offline [13], or at load-time [14]. These diversity-based defenses incur comparatively lower overhead and offer statistical guarantees against code-reyse exploits, but unlike FUZZBUSTER, they do not protect the program from the fault itself.

III. Background: FUZZBUSTER Adaptive Cybersecurity

FUZZBUSTER automatically tests and adapts multiple programs on a host machine by monitoring and adapting the programs' input and output signals. Consequently, programs defended by FUZZBUSTER may be compiled from any high-level programming language or interpreted by a virtual machine, provided the program or virtual machine emits fault signals (e.g., segmentation faults). FUZZBUSTER is designed to accomplish three general goals:

1) *Discovery*: proactively find vulnerabilities within the host's applications.
2) *Refinement*: produce a general, accurate, profile of every vulnerability that FUZZBUSTER encounters.
3) *Adaptation*: provided a refined vulnerability profile, create and assess an adaptation (i.e., patch or filter), and apply it if it improves the state of the application.

FUZZBUSTER is a fully automated system, and it uses a threat-based control strategy [4] to orchestrate its tools (see Table I) in pursuit of these goals.

When FUZZBUSTER discovers a fault in an application— or when it is notified of a *reactive* fault triggered by some other input source— it represents the fault as an *exemplar* that contains information about the system's state when it faulted, as shown in Figure 1. Note that FUZZBUSTER is not responsible for fault detection; we assume that other security and correctness mechanisms detect the fault and notify FUZZBUSTER.

An exemplar includes information for replicating the fault, such as environment variables and data passed as input to the faulting application (e.g., via sockets or `stdin`). Some of this data may be unrelated to the underlying vulnerability. For instance, when FUZZBUSTER encounters a fault during the Apache web server experiment discussed in Section VII, it captures all environment variables (none of which are necessary to replicate the fault), and the entire string of network input that was sent to the application (most of which is not necessary to replicate the fault). FUZZBUSTER uses fuzz-testing tools 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 to orchestrate these tools [4].

TABLE I
FUZZBUSTER'S TASKS. (* = NEW CONTRIBUTION)

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 [15], [16].
- `cross-fuzz`: Use Javascript and the DOM to fuzz-test web browsers.
- `wfuzz`: Fuzz-test web servers with templated attacks.
- `retrospective-fault-analysis`: Run faulting test cases through input filters to generate new faulting test cases.*

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.
- `crest`: Given source code, use concolic search to find constraints on the faulting input [17].
- `replace-all-chars`: Replace characters to generalize buffer overflows.*
- `replace-delimited-chars`: Replace delimited characters to generalize embedded buffer overflows.*
- `replace-individual-chars`: Replace single characters to generalize a faulting input pattern.*
- `insert-chars`: Insert characters to generalize regular expressions.*
- `shorten-regex`: Shorten wildcard patterns to find more accurate buffer overflow thresholds.*

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 [18] to evolve a new non-faulting program source and binary.

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. While refining the Apache web server vulnerabilities, FUZZBUSTER uses an environment variable fuzzer to test and remove unnecessary environment variables for replicating the fault, uses input fuzzers to delimit, test, and remove/replace unnecessary network input, and thereby develops a more accurate vulnerability profile.

FUZZBUSTER has several general adaptation methods, 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. In the case of adapting the Apache web server in Section VII, FUZZBUSTER creates input filters based on its vulnerability profiles: it extracts regular expressions that characterize the pattern of faulting inputs, including necessary character sequences (e.g., "Cookie:"), length-dependent wildcards (e.g., ".{256,}?"), and more. FUZZBUSTER then uses these input filters to identify potentially-faulting inputs and then discard them or rectify them, based on the application under test.

To date, our previous work on FUZZBUSTER has described the overall defense framework and integrated tools [19], [2], extended these tools with concolic testing [20], developed a meta-control strategy for mission- and time-sensitive orchestration of proactive and reactive tools [4], and improved the adaptation metrics [3]. This paper extends our previous publications with (1) new strategies for representing and generalizing vulnerabilities within program inputs, (2) new methods for assessing the safety and functionality of program adaptations, (3) new methods for injecting faults in production-level binaries, and (4) empirical results of FUZZBUSTER adaptating real programs with real vulnerabilities, illustrating the practical benefits of FUZZBUSTER's extensions.

IV. ASSESSING ADAPTATIONS

FUZZBUSTER cannot blindly apply adaptations, since they might have a negative impact on functionality or, even worse, they could create new faults altogether. Thus, FUZZBUSTER uses concrete metrics to assess the impact of candidate adaptations on security and functionality. In this section, we discuss FUZZBUSTER's adaptation metrics, and then we describe and evaluate two of FUZZBUSTER's strategies for assessing adaptations.

A. Metrics for Adaptive Cybersecurity

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). FUZZBUSTER automatically runs test cases to measure the safety and functionality of the programs it defends. 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 the following sets of test cases for each application under its control:

1) *Non-faulting test cases* are test cases that were supplied with an application for regression testing. FUZZBUSTER tracks which of these have correct behavior (i.e., output and return code), and which have different/incorrect behavior, given some adaptations.
2) *Faulting test cases* include exemplars that caused faults on their first encounter, and other faulting test cases encountered while refining the exemplar. FUZZBUSTER tracks which of these have been fixed by the adaptations

created so far, and which are still faulting. There are two specific types of faulting test cases:

a) *Reactive faulting test cases*: encountered by host notification and subsequent refinement (see Figure 1). These pose more of a threat, since the underlying vulnerability may have been triggered deliberately by an adversary.
b) *Proactive faulting test cases*: encountered by discovery and refinement (see Figure 1). These pose less threat, since they were discovered internally and FUZZBUSTER has no evidence that an adversary is aware of them.

FUZZBUSTER calculates two important metrics from these sets of test cases over time:

1) *Exposure* is computed as the number of unfixed faulting test cases over time. This represents an estimated window of exploitability.
2) *Functionality loss* is computed as the number of incorrect non-faulting test cases over time. This represents the usability that FUZZBUSTER has sacrificed for the sake of security.

Since FUZZBUSTER relies on test cases for measuring exposure and functionality, these measurements are only as complete as the set of faulting and non-faulting test cases, respectively. Before FUZZBUSTER has discovered faults or been notified of faults, there are no faulting test cases for any application. As FUZZBUSTER encounters proactive and reactive faults and refines those faults (e.g., by experimenting with different inputs), it will create additional faulting and non-faulting test cases. FUZZBUSTER accumulates these test cases— as well as any regression tests supplied with the application— and automatically runs them as described below to assess potential adaptations. FUZZBUSTER then applies and removes adaptations to fix the faulting test cases and restore the behavior of non-faulting test cases. These adaptations ultimately protect the host against adversaries.

We note that concolic testing tools (e.g., [21], [22]), including those already integrated with FUZZBUSTER [17], [20], can generate sets of test cases from the program's source code; however, since this work focuses on black-box fuzz-testing, FUZZBUSTER uses black-box tools to generate its test cases.

B. Two Adaptation Policies

Not all of FUZZBUSTER's adaptations improve the status of the analyzed program: some adaptations may sacrifice functionality (i.e., change the behavior of non-faulting test cases) without improving exposure (i.e., fixing faulting test cases), others may cause new faults, and still others may have no measurable effect. Using the metrics described above, FUZZBUSTER can apply different adaptation policies to assess whether an adaptation should be applied to a program.

In our first experiment, we compare two different adaptation policies. We describe these policies next, and then provide empirical results to characterize their effect on exposure and functionality.

1) Strict policy: In strict mode, FUZZBUSTER can never change the behavior of a non-faulting test case when adapting a program. What's more, all faulting test cases that correspond to the present vulnerability must be repaired (i.e., test cases corresponding to another vulnerability may still fault). The strict policy thereby only allows adaptations that are complete repairs and that preserve all known functionality, as determined by the available test cases. Once an adaptation is applied, it is never removed.

2) Relaxed policy: In relaxed mode, FUZZBUSTER may sacrifice functionality to fix faulting test cases. The exact balance can be tuned for different applications, but FUZZBUSTER's default priorities are:

1) Fixing reactive faulting test cases.
2) Fixing proactive faulting test cases.
3) Maintaining the behavior of non-faulting test cases.

This means that FUZZBUSTER will tolerate functionality loss (i.e., by changing the behavior of non-faulting test cases) in order to decrease exposure.

C. Experiment: Comparing Adaptation Policies

We conducted an experiment to compare the strict and relaxed adaptation policies. 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 non-faulting test cases set. These were gathered from examples in the `dc` manual and the Wikipedia `dc` entry, with the modulo and base conversion test cases removed.

Results are shown as functionality/exposure plots in Figure 2 and Figure 3. These plots display the following adaptive cybersecurity metrics, as described above:

- The number of faulting test cases FUZZBUSTER has identified through discovery and refinement (solid light red line).
- The number of those faulting test cases that FUZZBUSTER has fixed (dashed light red line).
- Exposure to vulnerabilities (area between light red lines), which FUZZBUSTER should ideally minimize.
- The number of non-faulting test cases FUZZBUSTER has for the application (solid dark blue line).
- The number of those non-faulting test cases whose return code and output behavior is preserved in the patched version (dashed dark blue line).
- Loss of functionality (area between dark blue lines), which FUZZBUSTER should ideally minimize.
- The patches that have been applied.

Figure 2 shows the results of FUZZBUSTER's strict policy. By definition, the strict policy preserves all functionality of

the application, 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. Following the patch are three numbers: (1) number of faulting test cases *created* by the patch; (2) number of faulting test cases *fixed* by the patch; and (3) number of non-faulting test cases *preserved* by the patch. As shown in Figure 2, under the strict policy, FUZZBUSTER created over 109 adaptations in 45 minutes, but only 11 passed the strict assessment criteria and were subsequently applied.

Figure 3 shows the results of the relaxed policy. This is plotted in the same way as the strict results, except we also include the dark blue non-faulting dataset. FUZZBUSTER accumulates non-faulting test cases over time by using fuzz-tools to generate and run test cases that do not produce faults.

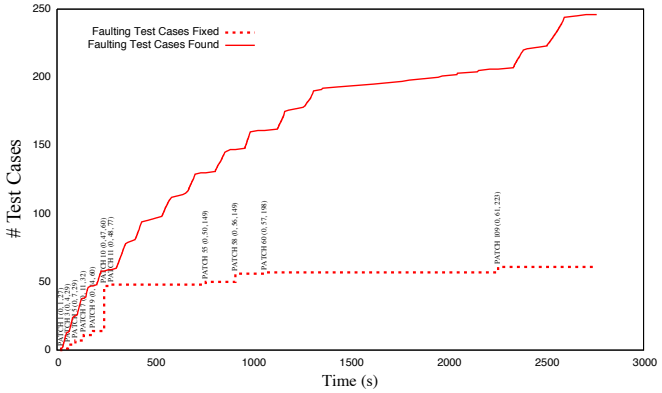

Fig. 2. FUZZBUSTER uses the strict policy to preserve its applications' functionality throughout the course of protective adaptation.

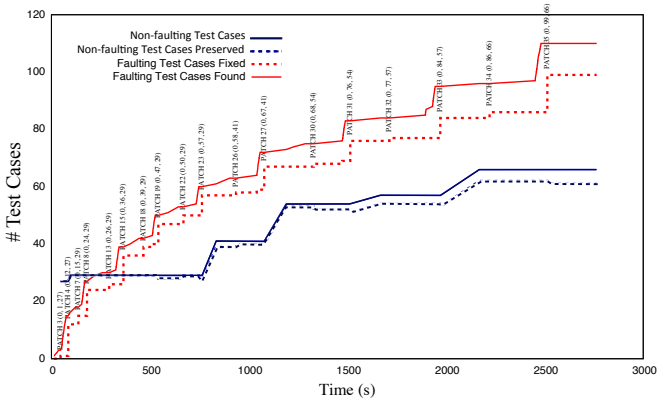

Fig. 3. FUZZBUSTER uses the relaxed policy to sacrifice functionality for the sake of improving security.

As shown in Figure 3, FUZZBUSTER incurs a loss of functionality (gap in between dark blue lines) to reduce its exposure to vulnerabilities. More specifically, it sacrifices up to 10% of its non-faulting test cases to fix faulting test cases, and it restores the behavior of erroneous 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 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 and assessing unused adaptations.
- At the end of each run, relaxed mode yields 110 total faulting test cases, and strict mode yields 245 total faulting test cases. This is because FUZZBUSTER was unable to apply as many adaptations in strict mode, so it discovered more variations of similar faulting test cases (which we verified were due to the same underlying vulnerability).

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 next describe new fuzz-tools that help close this margin and improve the effectiveness of FUZZBUSTER, as measured by the above adaptive cybersecurity metrics. All of the experiments in the remainder of this paper use the relaxed policy for assessing and applying adaptations.

V. TOOLS FOR DISCOVERY & REFINEMENT

In this section, we describe the new tools we developed to improve FUZZBUSTER's discovery and refinement capabilities. For the sake of comparison, we first review the set of fuzz tools we used in previous work [2], [3], [4].

A. Previous Fuzz-Tools

FUZZBUSTER's fuzz-tools included a random string generator for discovering faults (called `Fuzz-2001`) [15], [16] and various minimization (i.e., unnecessary character removal) tools for refining faults.

`Fuzz-2001` quickly constructs a sequence of printable and non-printable characters and feeds it as input to the program under test. This is effective for discovering some buffer overflows, problems with escape characters, and other such problems.

The minimization tools FUZZBUSTER uses to refine vulnerabilities include:

- *smallify*: semi-randomly removes single characters from the input string.
- *line-relev*: semi-randomly removes entire lines from the input string.
- *divide-and-conquer*: Use a binary search to attempt to remove entire portions of the input string.

Each of these tools is designed take a faulting test case as input, and produce smaller faulting test case(s).

Minimization tools can operate in a black-box fashion, where FUZZBUSTER does not have the source code or even access to the binary, since they only require an output signal to determine whether the program faulted.

B. New Fuzz-Tools

We now discuss several new tools that we have incorporated into FUZZBUSTER for discovering and refining faults. We then present empirical results comparing the new and existing tools to characterize the effects on the host's exposure to vulnerabilities.

These tools work with *input filter adaptations*; that is, program adaptations that remove content from input data before passing the data to the original program.

1) Retrospective Fault Analysis: We implemented and tested *Retrospective Fault Analysis (RFA)*, a new tool for vulnerability discovery. RFA works by finding the most recent faulting test case such that:

- The test case's input is filtered by the most recent adaptation applied, so some input data has been removed.
- The test case still faults, despite its input being filtered.

RFA then uses the test case— with filtered input— as an exemplar. This effectively allows FUZZBUSTER to fix test cases that still fault, despite incremental adaptations.

To illustrate why this is important, consider the following simplified example, where a program faults if it receives either `CRASH` or `fault` in an incoming message. Some messages may have more than one fault within them, e.g.:

- `Cookie: foo=...CRASH...fault...`
- `Cookie: foo=...faCRASHult...`

This means that FUZZBUSTER can automatically build a filter adaptation to address `CRASH`, but in both of the above cases, there will still be a `fault`. Using RFA, FUZZBUSTER will follow its `CRASH` adaptation with a retrospective investigation of the remaining `fault` test case(s). This produces a more complete analysis of problematic inputs, and it reduces the host's exposure to vulnerabilities, as we demonstrate in the experiments in Section VII.

2) Input Generalization Tools: As described above, minimization tools remove unnecessary characters for a fault. Unfortunately, refining vulnerabilities based on removal alone will tend to produce overspecific adaptations.

Consider the example of IP addresses within a packet header: minimization tools might trim 192.168.0.1 to 2.8.0.1, which might still produce the fault; however, an adaptation based on this model will only be effective when 2, 8, 0, and 1 are all present in the address.

FUZZBUSTER's new generalization tools go the extra step of replacing characters and inserting characters to generalize FUZZBUSTER's regular expression model of the faulting input pattern. This means that FUZZBUSTER will be able to substitute the IP address' digits with other digits to develop a more general, accurate adaptation.

We have implemented the following generalization tools:

- `replace-all-chars`: replaces all characters with different characters, reruns the test case, and then generalizes. This helps determine whether the test case is an instance of a buffer overflow. For example:

  `ABCDEFGH ==> .{8,}`

- `replace-delimited-chars`: splits the input into chunks, using common delimiters, removes and replaces delimited chunks, and then generalizes. For example:

  `host: 1.1.1.1\nCookie ==> .{0,}?Cookie`

- `replace-individual-chars`: removes and replaces individual characters, sensitive to character classes (e.g., letters, digits, whitespace, etc.), and generalizes. For example:

  `GCOJR34A59S94H ==> .*C.*R.*A.*S.*H`

- `insert-chars`: inserts characters in-between consecutive concrete characters, to test and relax adjacency constraints. For example:

  `CRASH ==> .*C.*R.*A.*S.*H`

- `shorten-regex`: reduces character counts within wildcard blocks to provide more accurate buffer overflow thresholds. For example:

  `host: .{951,} ==> host: .{256,}`

We conducted experiments on multiple programs to characterize the effect of generalization tools and RFA. We discuss these experiments and results next.

VI. EVOLVING FAULTY BINARIES FOR TESTING

An adaptive cybersecurity evaluation requires a set of programs to adapt and protect. Adapting production-grade software against known Common Vulnerabilities and Exposures (CVEs) is important for demonstrating realism (see Section VII-C); however, these CVEs do not always cover interesting spaces of program inputs for sensitivity analyses. Conversely, hand-injecting faults into the source code of production-grade software allows us to cover interesting input spaces; however, we aim to avoid hand-tailoring our dataset wherever possible.

Given these considerations, we designed and implemented a system to automatically inject faults into existing production-quality binaries using evolutionary programming. Our fault-injection approach takes the following inputs:

- The C source code of the application.
- A set of non-faulting test cases, where each test case is labeled *positive* or *negative*, and there is at least one test case with each label.

Given these inputs, our fault-injector wraps the positive test cases with shell code to return normally (i.e., 0) if there is no error (i.e., `[$? -lt 128 ]`), and then modifies negative test cases to return *abnormally* (i.e., 1) if there is no error. This transformation modifies the test cases to expect an error for any test cases labeled negative.

Our fault-injector then uses the application source code and transformed test cases as input to GenProg [23], an evolutionary program repair tool. In its normal mode of operation, GenProg generates variants of the program and uses the supplied test cases as a fitness function to select variants for the next round of mutation. By transforming the supplied test cases to expect failure from a subset of working test cases, we effectively make GenProg inject faults someplace in the

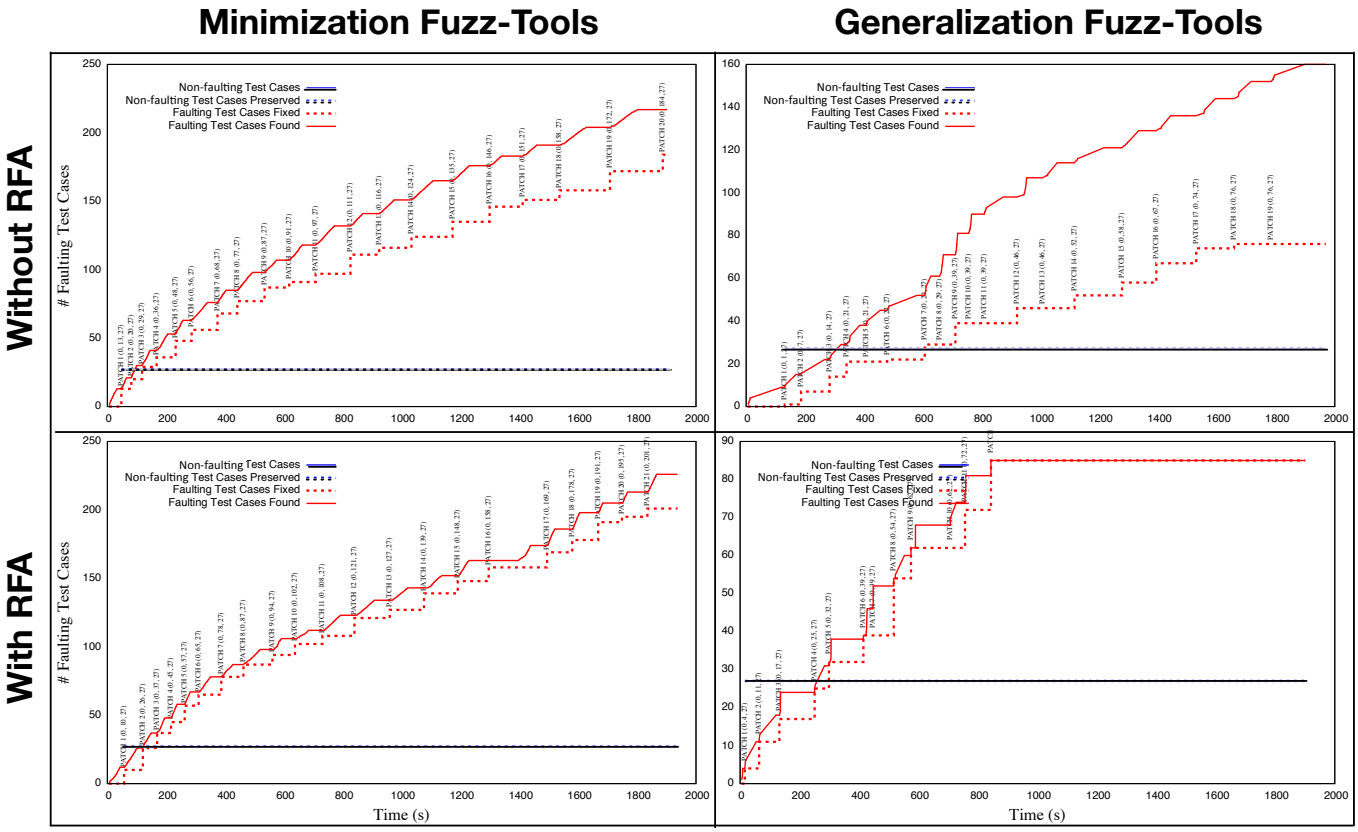

Fig. 4. Results comparing the exposure window of Retrospective Fault Analysis and minimization vs. generalization tools.

code to produce faults for the negatively-labeled cases, and still retain a working program for the positively-labeled cases.

Importantly, evolutionary fault-injection does not produce faults that are limited to the negatively-labeled test cases. Consider the case of the faulty `dc` used in Section IV-C: we supplied a negative test case that used the modulo (%) operation, and a dozen positive test cases that use other arithmetic operations. Evolutionary programming injected a null pointer dereference within an arithmetic helper function used for computing remainders, converting bases, and printing output with a non-decimal radix. Further, for all of these operations to fault, the internal calculator stack had to be non-empty. This demonstrates that evolutionary fault-injection will (1) produce non-trivial faults from simple sets of working test cases and (2) inject faults at arbitrary locations of the program, provided they produce a fault on the given input(s).

We used this automatic fault-injection method to create faulty binaries from the source code of unix command-line applications including `dc`, `fold`, `uniq`, and `wc`. Some of the fault-injected application variants faulted on the vast majority of possible inputs, and gracefully-terminating inputs were very rare. We discarded these variants since they lacked a stable, non-faulting operating mode.

We describe FUZZBUSTER's results adapting 16 fault-injected applications in Section VII-D.

VII. FUZZ-TOOL EXPERIMENTS

We conducted an empirical evaluation on different programs to measure the effect of RFA and the new generalization fuzz-tools. We divide this into four discussions: (1) a comparative analysis of minimization, generalization, and RFA on a single program; (2) an example of FUZZBUSTER sacrificing functionality in order to increase security; (3) a quantitative comparison of minimization and generalization using FUZZBUSTER to shield a web server against known vulnerabilities; and (4) adaptation statistics across multiple programs using FUZZBUSTER with generalization and RFA.

A. Comparative Analysis: Generalization, Minimization, RFA

For this experiment, we used the same fault-injected version of `dc` as in Section IV-C, with a faulty modulo operation. We ran FUZZBUSTER in five settings: with and without RFA, with either minimization or generalization tools (Figure 4); and then with RFA using *both* minimization and generalization tools (Figure 5).

The comparison plots in Figure 4 illustrate the tradeoffs of generalization and RFA. Minimization tools (Figure 4, left) quickly produce overspecific patches. For instance, **PATCH 16** in Figure 4 upper-left plot filters the pattern `.*9.*5.*%.*`. While this is a legitimate example of the fault, it does not characterize the fault in its entirety. By comparison, the generalization patches are slightly more general: **PATCH 6**

in the Figure 4 upper-right plot filters the pattern `.*d.*%.*` (where `d` duplicates the value on the stack).

Figure 4 also illustrates the effect of retrospective fault analysis. In the RFA trials, the exposure (the area between the red lines) is significantly reduced. This is because FUZZBUSTER often deploys a filter that addresses some – but not all – problems in a faulting input, and then RFA allows FUZZBUSTER to focus on the remainder of the problematic input. For instance, if a single test case has both a modulo operation and a base conversion, filtering out only one of these operations will not repair the test case.

In the setting with both generalization and RFA, FUZZBUSTER filters against the entire vulnerability within 15 minutes; in the other cases, FUZZBUSTER does not level off for over three hours.

Note that in all settings in Figure 4, FUZZBUSTER did not lose functionality of the underlying application, as measured by the correctness of the non-faulting test cases.

Figure 5 shows the results of FUZZBUSTER with both minimization and generalization enabled. It fixes the entire vulnerability and levels off in 18 minutes. Compared to the same condition with minimization disabled (Figure 4, lower right), enabling minimization made FUZZBUSTER spend some unnecessary time attempting to shorten faulting inputs when its generalization tools can find (or have already found) a more general representation of a faulting input pattern.

Also note that the minimization and generalization condition (Figure 5) destroys the functionality of one of the non-faulting test cases with its **PATCH 5**. The overgeneral **PATCH 5** filters out all occurrences of `z` (push the stack depth onto the calculator stack), which was indeed a character of a faulting input, but when FUZZBUSTER attempted to build a regular expression over its minimization and generalization results, the expression was overgeneral.

These results suggest that FUZZBUSTER's new input-generalizing tools are a suitable replacement for its input-minimizing tools.

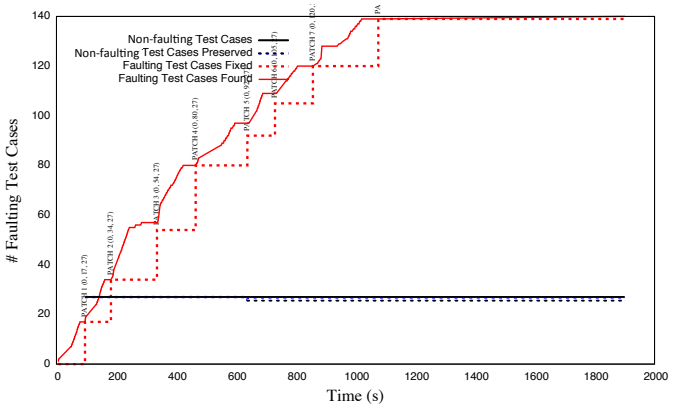

Fig. 5. Results using RFA, minimization, and generalization.

B. Sacrificing Functionality to Increase Security

We ran another FUZZBUSTER trial on a different fault-injected version of the `dc` binary. This version faulted whenever an arithmetic operation is invoked on an empty stack, so for instance, the sequence `''9 5 +''` would not fault, but the inputs `''+''` or `''4 n +''` would fault due to an empty stack (`''n''` pops the stack).

The results are shown in Figure 6. Using generalization tools and RFA, FUZZBUSTER isolates individual arithmetic operations and generates filters for each, ultimately disabling its arithmetic operations to prevent any faults. Note that almost every adaptation has an adverse impact on program functionality, but by design, these are acceptable losses to increase safety of the host.

TABLE II
FUZZBUSTER'S REACTION TIME ON CVES OF THE APACHE WEB SERVER.

CVE	RT (Min.)	RT (Gen.)	Speedup
2011-3192	96	4	24x
2011-3368-1	53	10	5x
2011-3368-2	32	10	3x
2011-3368-3	77	11	7x
2012-0021	36	3	12x
2012-0053	30	7	4x

Reaction times reported in minutes; speedup reported as quotient.

C. Adapting a Web Server

We conducted FUZZBUSTER experiments on known Common Vulnerabilities and Exposures (CVEs) on the Apache web server. This demonstrates FUZZBUSTER working on larger production-quality applications with real vulnerabilities, and it shows the generality of FUZZBUSTER and its fuzz-tools.

For each trial, we initialized FUZZBUSTER with the Apache web server as the only application under test. We then sent a faulting message to the server— as dictated by the corresponding CVE— and FUZZBUSTER detected the reactive fault and began its fuzzing. Table II reports how many minutes FUZZBUSTER took to produce an input filter adaptation (from experiment start to patch time) for the corresponding CVE using only minimization tools (i.e., "Min."), only generalization tools (i.e., "Gen."), and the speedup provided by generalization tools.

In addition to producing more general patches, the generalization tools also yield a significant speedup factor between 3x and 24x, and on average, produce useful adaptations in an order of magnitude less time.

For these CVE trials, RFA was not necessary since FUZZBUSTER fixes all faulting test cases with the first patch it produces.

D. Statistics Across Programs

We now present additional results from using FUZZBUSTER with the generalization tools and retrospective fault analysis on 16 fault-injected binaries created using the process described in Section VI.

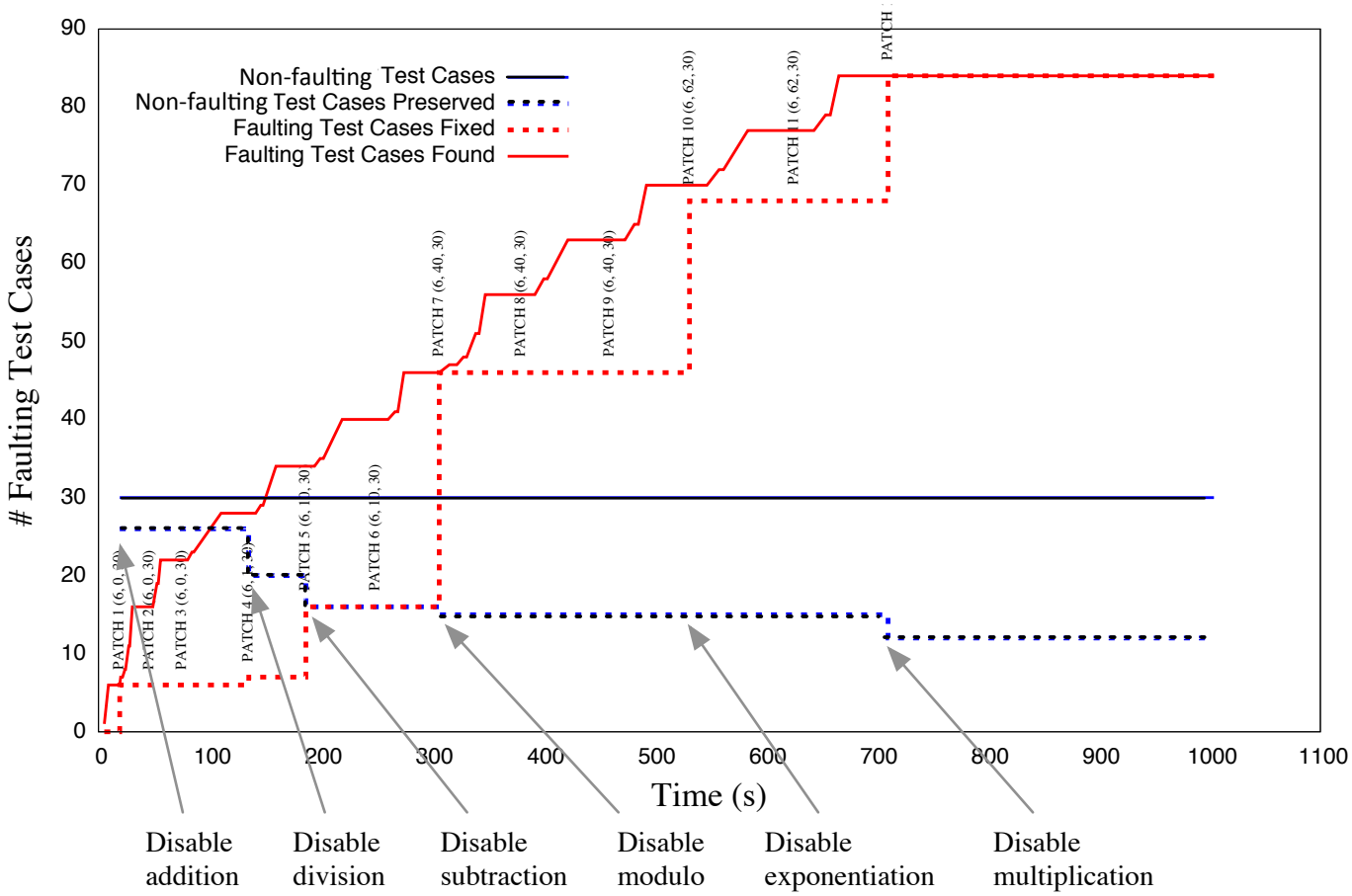

Fig. 6. FUZZBUSTER sacrifices functionality to protect the program against vulnerabilities.

FUZZBUSTER automatically analyzed each faulty binary for two hours, using a mix of proactive fuzz-tools (e.g., `Fuzz-2001` and `RFA`), refinement fuzz tools (e.g., the generalization fuzz-tools), and adaptation strategies (e.g., input filters).

Fuzzing *leveled off* (i.e., FUZZBUSTER patched the entire injected fault, based on our manual analysis of patches) on 10/16 binaries. Of these leveled-off binaries, FUZZBUSTER took an average of 5.87 minutes to level off, and it sacrificed an average of 6% functionality (i.e., by changing the output of non-faulting test cases). FUZZBUSTER retained full functionality on 7 of the 10 leveled-off binaries.

Over all 16 fault-injected binaries, FUZZBUSTER created an average of 8.2 adaptations and applied an average of 7.8, which amounts to a 95% usage of the adaptations it created. Over all binaries, FUZZBUSTER fixed an average of 82% of the faulting test cases and sacrificed an average of 10% functionality during each 2-hour trial. This suggests that when FUZZBUSTER cannot generate a perfect adaptation, it still manages to close the exposure window over time.

VIII. CONCLUSIONS AND FUTURE WORK

FUZZBUSTER is designed to discover vulnerabilities and then quickly refine and adapt its applications to prevent them from being exploited by attackers. This paper uses FUZZBUSTER as a research tool to further the state-of-the-art in measuring and improving adaptive cybersecurity. We presented useful exposure and functionality metrics, we used these metrics to compare and evaluate FUZZBUSTER's self-adaptation policies, and we used the metrics to characterize the benefits FUZZBUSTER's new fuzz-testing tools— retrospective fault analysis and input generalization— aimed at improving the quality and efficiency of adaptive cybersecurity. Further, we described how to automatically inject faults into production-grade software to build datasets for adaptive cybersecurity experimentation.

We presented empirical results of FUZZBUSTER's automated analysis of both fault-injected programs and real CVEs, comparing the vulnerability exposure, functional loss, and reaction time. When analyzing fault-injected programs, the generalization fuzz-tools and RFA reduced vulnerability exposure by a factor of five on fault injected programs, and allowed FUZZBUSTER to shield more of the vulnerability in less time. When analyzing the Apache HTTP server, the new fault generalization tools yielded an order of magnitude speedup in reaction time over the previous fault minimization tools.

Currently, FUZZBUSTER uses a wrapper around the pro-

grams it controls, and its wrapper filters all incoming data according to the current adaptations (e.g., input filters) before sending the data to the binary. One next step is to revise the program's binary directly, and embed the input filters as preprocessors.

The generalization fuzz-tools and RFA are all domain-independent strategies, and we demonstrated this by using them to improve program analysis on command-line filter programs (e.g., `wc`), state-dependent standard input programs (e.g., `dc`), and grammar-specific web programs (e.g., Apache HTTP server). The most domain-specific enhancement is the `replace-delimited-chars` tool that uses common delimiters to analyze portions of data. This tool contributed significantly to the speedup of FUZZBUSTER's analysis of HTTP headers in the Apache HTTP server experiment. We believe that we will see additional performance benefits by adding more domain-specific knowledge to FUZZBUSTER, including input grammars (e.g., packet header structure) and deeper application models (e.g., recording application command-line options and values).

Our fault-injector currently requires source code in order to use GenProg [23]. We are investigating fault-injection using evolutionary programming methods that operate directly on assembly code or compiled binaries (e.g., [24]), and do not require source code. This has the advantages of (1) being high-level-language-independent and (2) producing more diverse vulnerabilities at the binary or library level.

We anticipate using the adaptive cybersecurity metrics from this paper to evaluate future design decisions for FUZZBUSTER and other adaptive cybersecurity projects.

ACKNOWLEDGMENTS

This work was supported by The Defense Advanced Research Projects Agency (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] D. J. Musliner, S. E. Friedman, and J. M. Rye, "Automated fault analysis and filter generation for adaptive cybersecurity," in Proceedings of ADAPTIVE 2014: The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, June 2014, pp. 56–62.

[2] D. J. Musliner, J. M. Rye, D. Thomsen, D. D. McDonald, and M. H. Burstein, "FUZZBUSTER: A system for self-adaptive immunity from cyber threats," in Eighth International Conference on Autonomic and Autonomous Systems (ICAS-12), March 2012, pp. 118–123.

[3] D. J. Musliner et al., "Self-adaptation metrics for active cybersecurity," in SASO-13: Adaptive Host and Network Security Workshop at the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems, September 2013, pp. 53–58.

[4] D. J. Musliner, S. E. Friedman, J. M. Rye, and T. Marble, "Meta-control for adaptive cybersecurity in FUZZBUSTER," in Proc. IEEE Int'l Conf. on Self-Adaptive and Self-Organizing Systems, September 2013, pp. 219–226.

[5] 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.

[6] 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.

[7] H. Shrobe et al., "AWDRAT: a cognitive middleware system for information survivability," AI Magazine, vol. 28, no. 3, 2007, p. 73.

[8] 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.

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

[10] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 571–585.

[11] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 40–51.

[12] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in Proceedings of the 2010 workshop on New security paradigms. ACM, 2010, pp. 7–16.

[13] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 601–615.

[14] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 157–168.

[15] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," Communications of the ACM, vol. 33, no. 12, 1990, pp. 32–44.

[16] B. Miller, G. Cooksey, and F. Moore, "An Empirical Study of the Robustness of MacOS Applications Using Random Testing," in Proceedings of the 1st international workshop on Random testing. ACM, 2006, pp. 46–54.

[17] 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

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

[19] 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, pp. 137–140.

[20] 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, pp. 9–14.

[21] P. Godefroid, M. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in Proceedings of the Network and Distributed System Security Symposium, 2008, pp. 151–166.

[22] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in Proceedings of the 8th USENIX conference on Operating systems design and implementation. USENIX Association, 2008, pp. 209–224.

[23] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," Software Engineering, International Conference on, 2009, pp. 364–374.

[24] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010, pp. 313–316.