Automated Fault Analysis and Filter Generation for Adaptive Cybersecurity

David J. Musliner, Scott E. Friedman, Jeffrey M. Rye
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {dmusliner,sfriedman,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. Adaptive cybersecurity involves efficiently improving software security to minimize the window of attack, and also preserving software functionality as much as possible. This paper presents new tools that have been integrated into FUZZBUSTER adaptive cybersecurity. These tools produce more general, accurate adaptations, increase the efficiency of FUZZBUSTER's diagnoses and adaptation operations, and preserve the software's functionality. We report the results of FUZZBUSTER's analysis of 16 fault-injected command-line binaries and six previously known bugs in the Apache web server. We compare results over different configurations of FUZZBUSTER to characterize the benefits of the new fuzz-testing tools.**

Keywords-cyber defense; automatic filter generation.

I. INTRODUCTION

Cyber-attackers constantly threaten today's computer systems, increasing the number of intrusions every year [1], [2]. 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. 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 make 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 window of exposure to vulnerability over time.

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 defend against them. We have implemented these strategies within the FUZZBUSTER integrated system for active cybersecurity [3], which includes metrics [4], and metacontrol [5] for self-adaptative software immunity. 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:

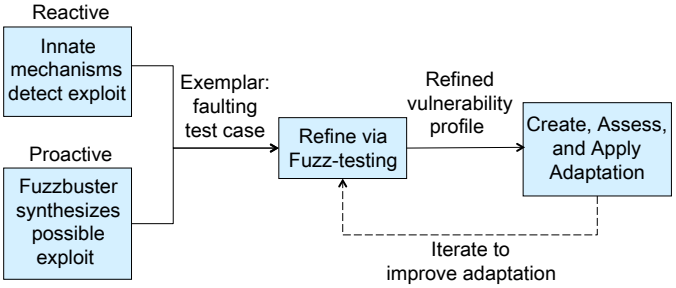

Fig. 1. FUZZBUSTER automatically finds vulnerabilities, refines its understanding of their extent, and creates adaptations to shield or repair them.

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.

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 FUZZBUSTER's process of discovering, refining, and repairing vulnerabilities, which motivates

our research on adaptation metrics. We then describe FUZZBUSTER's novel diagnosis tools for adaptive cybersecurity and we summarize the results of several experiments.

II. Background: FUZZBUSTER Active Cybersecurity

FUZZBUSTER tests and adapts multiple applications on a host machine. When FUZZBUSTER discovers a fault in one of these applications— 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 (e.g., the fault might be replicated without a specific environment variable binding). 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 [5] 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.

A. 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.

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 (reference) 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 caused 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.

We can calculate 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 (reference) test cases over time. This represents the usability that FUZZBUSTER has sacrificed for the sake of security.

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 accrue faulting test cases. FUZZBUSTER then applies and removes adaptations to fix these faulting test cases. These adaptations ultimately protect the host against adversaries.

FUZZBUSTER's assessment policy allows it to 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.

B. Pre-existing Tools for Discovery & Refinement

Since this paper presents new tools for discovery and refinement (Section III), for the sake of comparison we first review the set of fuzz tools we used in previous work [5], [3], [4]. Those tools included a random string generator for discovering faults (called `Fuzz-2001`) 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. All they require is an output signal to determine whether the program faulted.

III. New Discovery & Refinement 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.

Both of these tools work with *input filter adapations*; that is, program adaptations that remove content from input data before passing the data to the corresponding program.

A. 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 improves the host's exposure to vulnerabilities, as we demonstrate in our experiments.

B. Input Generalization Tools

As described in Section II-B, 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 determines 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 relax adjacency constraints. For example:

  `CRASH ==> .*C.*R.*A.*S.*H`
- `shorten-regex`: removes characters 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.

IV. Experiments

We conducted an empirical evaluation on different programs to measure the effect of RFA and the new generalization fuzztools. 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.

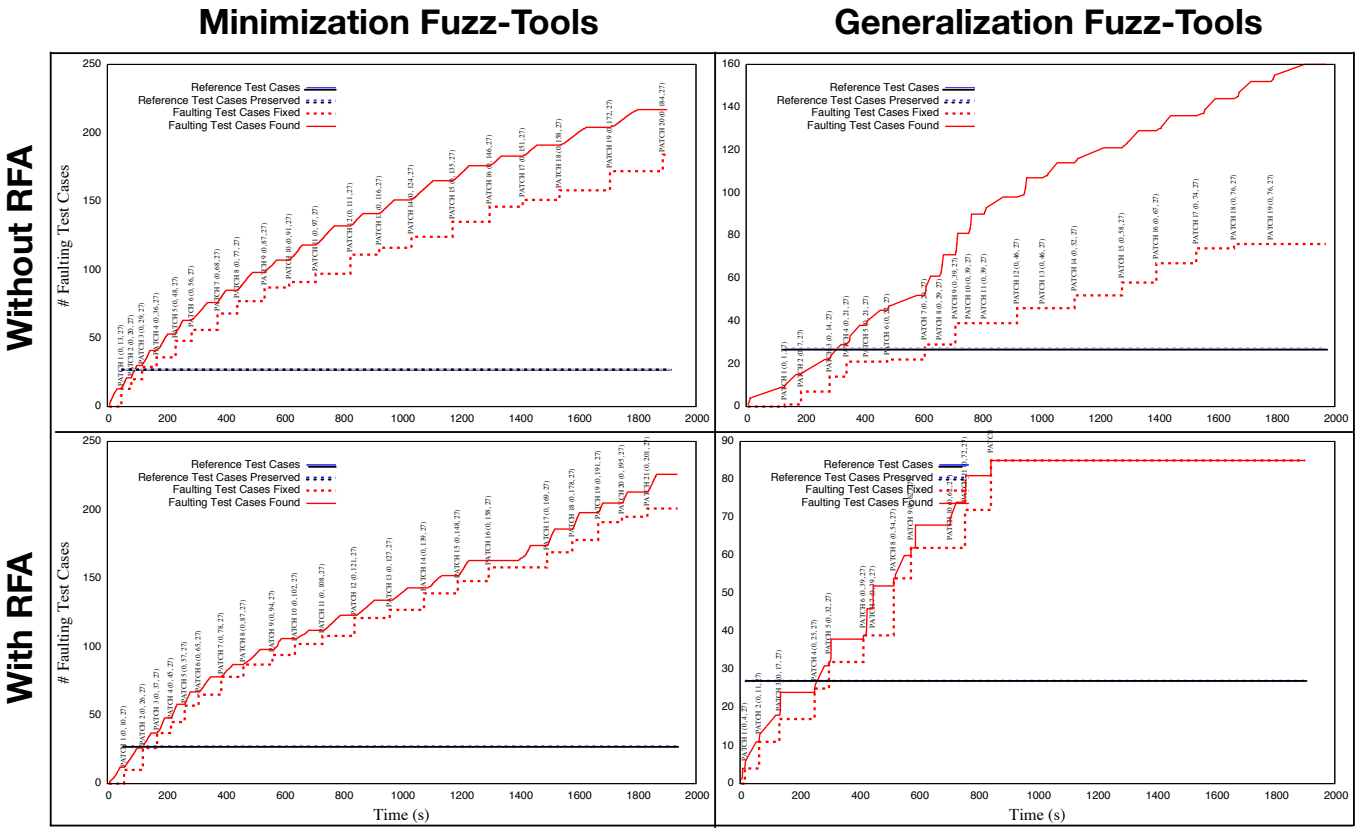

Fig. 2. Results comparing the exposure window of Retrospective Fault Analysis and minimization vs. generalization tools.

A. Comparative Analysis: Generalization, Minimization, RFA

For this experiment, we used a fault-injected version of `dc`, a unix-based, `stdin`-based desktop calculator program. The fault in `dc` was injected within the internal modulo (i.e., remainder) operation. This operation is reached by invoking the `%` command with at least two numbers on the stack, printing with a non-decimal output radix, changing the input radix, or invoking base conversion.

We ran FUZZBUSTER in five settings: with and without RFA, under either minimization or generalization tools (Figure 2); and then with RFA using *both* minimization and generalization tools (Figure 3).

Each of these plots display the following important data for adaptive cybersecurity:

- 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).
- The number of reference (non-faulting) test cases FUZZBUSTER has for the application (solid dark line).
- The number of those non-faulting test cases whose return code and output behavior is preserved in the patched version (dashed dark line).
- Loss of functionality (area between dark lines).

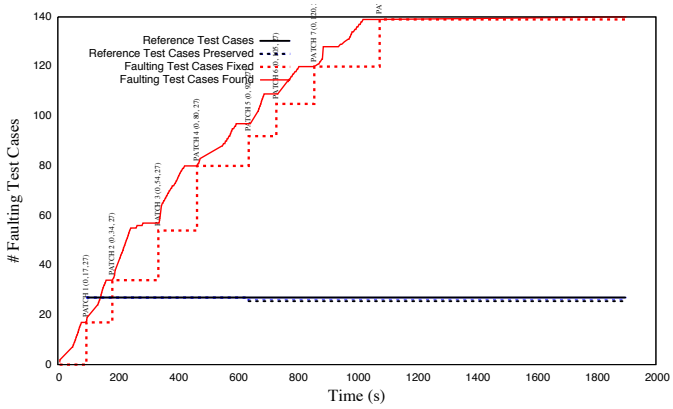

Fig. 3. Results using RFA, minimization, and generalization.

- The patches that have been applied.

The comparison plots in Figure 2 illustrate the trade-offs of generalization and RFA. Minimization tools (Figure 2, left) produce quick, overspecific patches. For instance, **PATCH 16** in the Figure 2 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.

Figure 2 also illustrates the effect of retrospective fault analysis. In the RFA trials, the exposure (distance between

the light 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 2, FUZZBUSTER did not lose functionality of the underlying application, as measured by the correctness of the reference test cases.

Figure 3 shows the results of FUZZBUSTER with both minimization and genralization enabled. It fixes the entire vulnerability and levels off in 18 minutes, but it also destroys the functionality of one of the reference test cases, since its **PATCH 5** was overgeneral.

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 (and ``n'' pops the stack).

The results are shown in Figure 4. 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.

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. Figure 5 reports how many minutes FUZZBUSTER took to produce an input filter adaptation 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.

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

Fig. 5. FUZZBUSTER's reaction time (from simulation start to patch time), using minimization and generalization tools on six known CVEs of the Apache web server. Reaction times are reported in minutes, and speedup is reported as the quotient.

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.

We used GenProg [6], an evolutionary program repair tool, to create faulty binaries from the source code of unix command-line applications including `dc`, `fold`, `uniq`, and `wc`. We achieved this by modifying the GenProg test cases— which GenProg uses as a fitness function— to expect a fault on certain inputs. This way, GenProg would generate selectively-faulting binaries based on our specifications.

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 reference 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.

V. RELATED WORK

As previously noted, the FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 applied to 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, 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.,

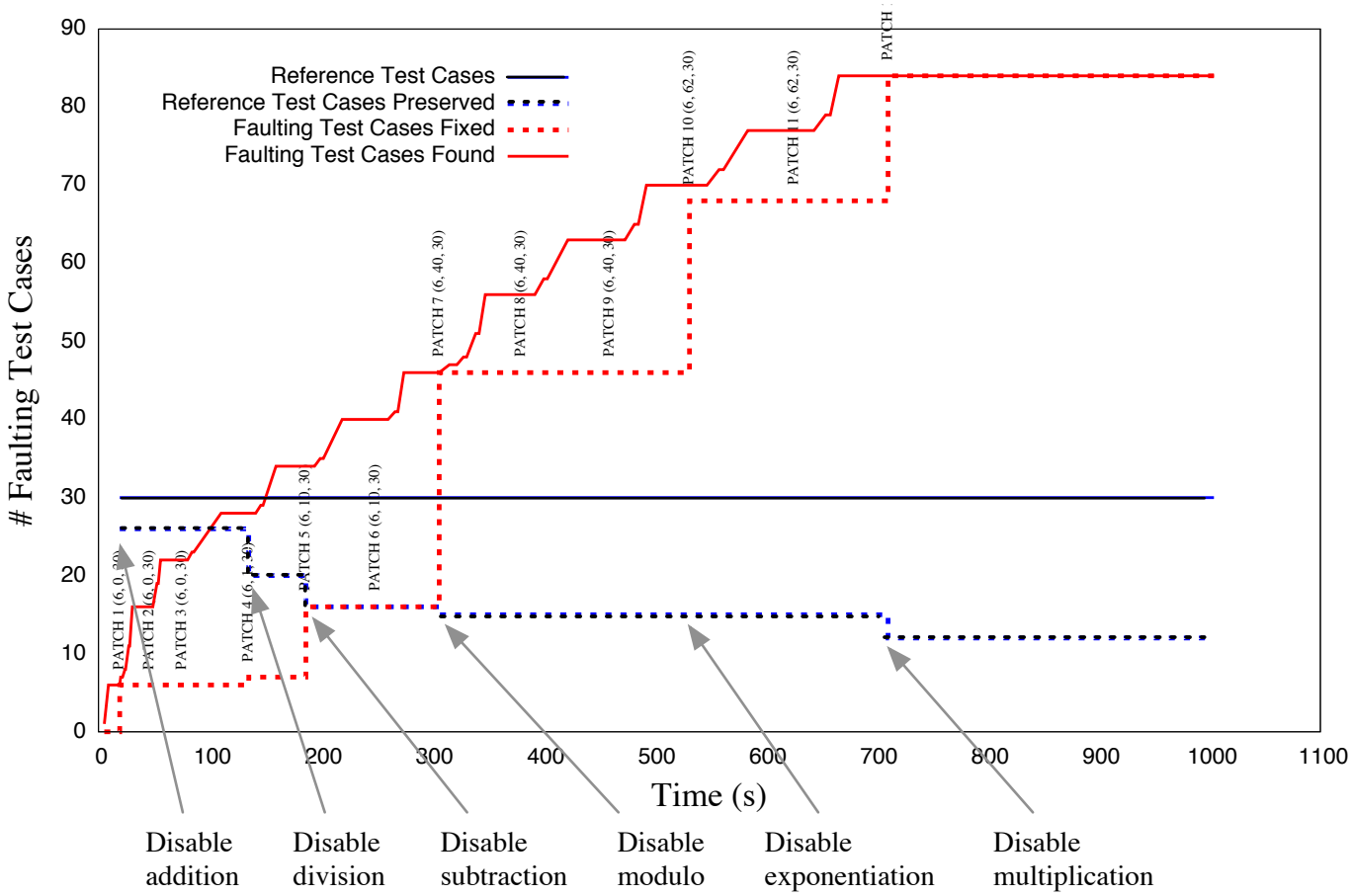

Fig. 4. FUZZBUSTER sacrifices functionality to protect the program against vulnerabilities.

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.

VI. CONCLUSION 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 presented two advances in FUZZBUSTER's tools — retrospective fault analysis and generalization fuzz-tools — aimed at improving the quality and efficiency of FUZZBUSTER's adaptations. We presented empirical results of FUZZBUSTER's automated analysis of fault-injected programs and real CVEs, using objective metrics for adaptive cybersecurity such as 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 filter out

more of the vulnerability in less time. When analyzing the Apache HTTP server, the fault generalization tools yielded an order of magnitude speedup in reaction time over the existing fault minimization tools.

At present, FUZZBUSTER uses a wrapper around the programs 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-general 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 structures to FUZZBUSTER, including input grammars (e.g., packet header structure) and deeper application models (e.g., recording application command-line options and values).

We anticipate using the adaptive cybersecurity metrics from this paper (see also [5], [4]) to evaluate future design decisions for FUZZBUSTER and other active cybersecurity projects.

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, May-June 2010, pp. 70 –73.

[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] 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.

[4] 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.

[5] 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, sep 2013.

[6] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," Software Engineering, International Conference on, vol. 0, 2009, pp. 364–374.

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

[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.