Meta-control for Adaptive Cybersecurity in FUZZBUSTER

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

***Abstract*—Modern cyber attackers use sophisticated, highly-automated vulnerability search and exploit development tools to find new ways to break into target computers. To protect against such threats, we are developing FUZZBUSTER, a host-based adaptive security system that automatically discovers faults in hosted applications and incrementally refines and repairs the underlying vulnerabilities. To perform this self-adaptation, FUZZBUSTER uses meta-control to coordinate a diverse and growing set of custom and off-the-shelf fuzz-testing tools. FUZZBUSTER's greedy meta-control strategy considers adaptation deadlines, the exploit potential of vulnerabilities, the usage schedule of vulnerable applications, and the expected performance of its various fuzz-testing and adaptation tools. In this paper, we demonstrate how FUZZBUSTER's meta-control reasons efficiently about these factors, managing task selection to maximize the system's safety and effectiveness.**

Keywords-self-adaptive immunity, cybersecurity, fuzz-testing, meta-control

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.

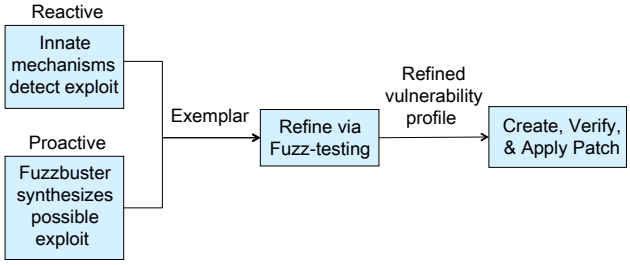

Figure 1. When reacting to an observed fault, FUZZBUSTER creates an exemplar that reflects the environment and inputs at the time of the fault. During proactive exploration, FUZZBUSTER synthesizes exemplars that could lead to a novel fault.

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.

This paper focuses on FUZZBUSTER's meta-control reasoning, which coordinates proactive and reactive adaptation under time constraints, without incurring excessive meta-control processing overhead. We begin by outlining FUZZBUSTER's process of discovering, refining, and repairing vulnerabilities, which motivates our research on efficient and effective meta-control. We then describe our approach and summarize the results of several experiments that demonstrate FUZZBUSTER's meta-control in action.

A. Discovering, Refining, and Repairing Threats Using FUZZBUSTER

Whether a fault is proactively discovered by FUZZBUSTER or reactively observed on the host, FUZZBUSTER represents the fault as an *exemplar* that contains information about the system's state when it faulted.

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

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.

exemplar, trying to characterize the minimal inputs needed to trigger the fault.

When the vulnerable application's source code is available, FUZZBUSTER can use concolic testing (a combination of concrete and symbolic execution) to find more general constraints on the inputs that produce the fault [6]. When no source code is available, FUZZBUSTER can perform black-box refinement by using stochastic and deterministic fuzz-testing tools to generalize the pattern of inputs that produce the fault. Refinement is an iterative process, where each concolic or black-box tool improves the *vulnerability profile* that FUZZBUSTER uses to characterize the vulnerability. When refinement is complete, the vulnerability profile characterizes the vulnerability as generally and accurately as FUZZBUSTER's toolset will allow.

The vulnerability profile ultimately guides FUZZBUSTER's adaptation strategy. FUZZBUSTER can deploy input filters, environment variable filters, or source-code repair and recompilation, protecting against entire classes of exploits that may be encountered in the future.

Unfortunately, processing power is limited and FUZZBUSTER is racing against cyber-attackers to discover and fix vulnerabilities before they are exploited. Consequently, FUZZBUSTER must repair vulnerabilities quickly, without spending excessive time deciding among tasks or executing unnecessary tasks.

In this paper, we describe how FUZZBUSTER's meta-control orchestrates the discovery, refinement, and elimination of software vulnerabilities. We begin by describing the CIRCA Adaptive Mission Planner [10] on which FUZZBUSTER's meta-control approach is built. We then describe how the concepts from CIRCA map to the FUZZBUSTER cybersecurity domain, and how we extended the general meta-control approach to support additional adaptation capabilities. We present experimental evidence that FUZZBUSTER's meta-control attends to important factors in adaptive cybersecurity, including application usage schedules, vulnerability threat levels, and adaptation deadlines.

II. BACKGROUND: CIRCA ADAPTIVE MISSION PLANNER

FUZZBUSTER's meta-control is built on elements from CIRCA's Adaptive Mission Planner (AMP). CIRCA (the Cooperative Intelligent Real-Time Control Architecture) is designed to control autonomous agents in mission-critical time-sensitive domains. CIRCA includes a Controller Synthesis Module (CSM) that automatically synthesizes reactive controllers. The CIRCA CSM is generally over-constrained, in that it cannot produce optimal plans for an entire mission in the allotted time. The CIRCA AMP's job is to actively manage the CSM, choosing which problems the CSM tries to solve at any given time. The AMP uses a greedy deliberation scheduling algorithm to (1) approximate the incremental expected utility of CSM-planning for all future mission phases, and then (2) initiate CSM-planning for the highest-utility phase.

The AMP's meta-level management relies on the following concepts that are relevant to FUZZBUSTER:

- *Mission phases* are a partitioning over time, from the beginning to the end of the agent's mission. Every mission phase has a start time and an end time. Since the AMP deliberates before and during the mission, it tracks the *current phase* in the mission.
- *Threats* are hazardous factors within a mission phase that could end the mission, such as equipment failure or adversaries. Threats have a numerical *lethality* value, where a higher lethality indicates a higher likelihood of failure if the threat is not addressed by a reactive plan.
- *Goals* are desirable factors within a mission phase. A plan that is expected to achieve a goal is assigned a numerical *reward*, where reward represents goal desirability.
- *Tasks* are deliberation actions that take non-negligible time (*e.g.*, running the CSM planner) and have some probability of success (*e.g.*, creating a plan that is

expected to successfully achieve all of the selected goals and prevent threat-induced failures).
- A *performance profile* for a task k contains estimates of its time duration $t(k)$ and its probability of success $P(k)$.

In CIRCA domains, each task involves planning within a single mission phase in order to achieve a set of goals and neutralize a set of threats. Some planning tasks can take longer than others, since some phases have more threats and goals to consider than others.

The AMP selects the next task to execute from a set of possible tasks by computing the *incremental utility* of each task (*i.e.*, its rate of utility increase over time) based on the threats and goals within the task's phase, and the future rewards from subsequent mission phases. We describe these equations top-down, starting with incremental utility $U(k)$, of a task k:

$$U(k) = \frac{P(k)(E(k) - E(\emptyset))}{t(k)} \quad (1)$$

This is based on the *expected future payoff* $E(k)$ of performing the task, and the expected future payoff $E(\emptyset)$ of doing nothing. We compute $E(k)$ as follows, where G_k and T_k are the goals and threats addressed by task k, phase p_i is the phase for task k, and the sequence of phases $p_j, \ldots, p_n$ is the sequence of future phases after p_i:

$$E(k) = R(G_i \cup G_k)S(T_i \setminus T_k)\sum_{p=j}^{n} R(G_p)\prod_{p'=j}^{p} S(T_{p'}) \quad (2)$$

We use $R(G)$ as the numerical reward for a set of goals G and $S(T)$ as the numerical survival probability for a set of threats T. The expected future payoff first computes the reward of the goals secured in the current phase G_i and the goals secured by the task k, G_k, multiplied by the survival probability of the threats in the phase T_i, but without the threats T_k addressed by the task. So far, this represents the expected reward— given the threats— within the phase of the task. Expected future payoff also includes the sum of future phase rewards, scaled by the cumulative survival probability of surviving through all preceding phases. This means that if the agent can increase survival probability in a phase (*i.e.*, by neutralizing threats) then it will increase the expected reward of all future phases and ultimately increase its expected future payoff.

In the case of $E(\emptyset)$, we set $G_k = \emptyset$ and $T_k = \emptyset$, so $E(k) - E(\emptyset)$ (from the incremental utility equation) is the marginal utility of performing task k.

The AMP computes these values to select the task with the maximum incremental utility and then execute that task.

Like CIRCA, FUZZBUSTER uses the AMP and the above equations for task selection, but the concepts from CIRCA (*e.g.*, phases, threats, and goals) must be mapped into the domain of cybersecurity, and we must add functionality to orchestrate the types of tasks shown in Table I.

III. FUZZBUSTER META-CONTROL

FUZZBUSTER must coordinate multiple types of tasks (see Table I) to simultaneously test and adapt multiple applications on its host.

The ultimate FUZZBUSTER objective is system security, which differs from traditional CIRCA missions, but we can define the AMP concepts from Section II to utilize the same utility, payoff, reward, and survival metrics.

The CIRCA concepts map to FUZZBUSTER as follows:

- Each *mission phase* is a nonzero duration of time where zero or more applications are used (*i.e.*, by the system or the end user).
- A *goal* is the use of an application within a phase, where the numerical reward is proportional to the length of the phase.
- A *threat* is either (a) a known vulnerability in an application, or (b) a potential vulnerability in an application. Known vulnerabilities have been found or replicated by FUZZBUSTER, and potential vulnerabilities have not (so they are orders of magnitude less lethal), but any application is potentially vulnerable. Known and potential vulnerabilities exist in every phase where the corresponding application is used.
- A *task* is any discovery, refinement, or adaptation task shown in Table I, for some application on the host.
- A *performance profile* for a task includes its runtime estimate as well as an estimate of its likelihood of contributing to a successful protective adaptation.

Note that application usage in a particular mission phase provides a reward, but also exposes the system to one or more threats that reduce the likelihood of future rewards. This is because using an application extends the attack surface of the host, and when that attack surface is vulnerable, it might be exploited.

FUZZBUSTER cannot alter the mission phases or the usage schedule of the applications therein, but it can select tasks that increase the probability of survival and thereby increase future reward, given the usage schedule. It must choose among proactive fuzz-testing tasks (*e.g.*, `fuzz-2001`) for different applications on the host, and it must choose among different types of refinement and adaptation tasks (*e.g.*, `smallify` and `find-regex`) for any identified vulnerability. FUZZBUSTER uses the incremental utility function from Section II to select the currently-executable task with the highest incremental utility. In effect, this means that FUZZBUSTER will select one task over another task if the following are true, all else being equal:

- The associated application is used more frequently. This helps FUZZBUSTER focus on active areas of the attack surface.

- The associated application will be used sooner. This helps FUZZBUSTER focus on more imminent potential exploits.
- The associated vulnerability (*e.g.*, of a refinement task) is more lethal. This helps FUZZBUSTER address more critical problems sooner.
- The task is more likely to succeed. This reduces the likelihood of FUZZBUSTER wasting cycles on futile tasks.
- The task is expected to take less time. This helps FUZZBUSTER improve the security of the system at the highest rate possible.

Unfortunately, adopting the original, unmodified AMP meta-control code also causes an undesirable behavior. Suppose that FUZZBUSTER is fuzz-testing two applications, one of which is used 90% as frequently as the other. All other things being equal, FUZZBUSTER will *only* select tasks for the more frequently used application, since it yields more utility. Since FUZZBUSTER cannot ignore slightly-less-used applications in its search for unknown vulnerabilities, we implemented a balancing mechanism, which we discuss next.

A. Balancing Effort Over the Attack Surface

FUZZBUSTER's meta-control aims to balance the time it spends fuzz-testing its applications according to the relative proportion of time each application will be used in future mission phases.

FUZZBUSTER embeds effort-balancing into its phase-based utility computation. For a given task k, it computes a balance coefficient b_i for each phase i. The coefficient b_i is computed using three vectors that track time across applications: (1) the present time allocation t over all applications; (2) the task's allocation t_k over all applications;[1] and (3) the time allocation t_i over applications in phase i. The balance coefficient is computed as follows, where $\hat{t}$ indicates a normalization of vector t:

$$b_i = 1 + (\hat{t_i} \cdot \widehat{t + t_k}) - (\hat{t_i} \cdot \hat{t}) \tag{3}$$

Phase-based application balancing causes FUZZBUSTER to allocate its fuzz-testing effort across applications according to their share of future usage, all else being equal. By incorporating this balancing scalar into the utility calculation, FUZZBUSTER is able to smoothly merge the desired balancing behavior with the other meta-control behaviors noted earlier. For example, as we will show in our experimental results, FUZZBUSTER will temporarily abandon its effort-balancing to fix high-impact known vulnerabilities. This is because addressing a known vulnerability (to greatly increase survival probability) provides orders of magnitude more utility than achieving balance over applications.

[1]At present, FUZZBUSTER tasks only address a single application, so vector t_k contains exactly one nonzero dimension.

B. Adaptation Deadlines

If FUZZBUSTER only adapted to a vulnerability once its corresponding vulnerability profile was fully refined, FUZZBUSTER would leave its host unprotected throughout a potentially lengthy refinement process. To quickly foil potential exploits of known vulnerabilities, FUZZBUSTER has an *adaptation deadline* parameter for each vulnerability, which tells FUZZBUSTER how long it has to make an initial adaptation (*e.g.*, apply an environment patch, apply an input filter, or evolve a new binary). Deadlines are optional parameters, but if a deadline is set, it is *pending* if the application has not been adapted for that specific vulnerability, and it is *satisfied* otherwise.

FUZZBUSTER attempts to satisfy pending deadlines by creating the best adaptation possible in the allotted time, and then it continues refining the vulnerability to achieve a more general and accurate final repair. FUZZBUSTER's meta-control addresses pending deadlines using the following procedure:

1) Calculate the time remaining before the deadline.
2) Calculate the *time-bounded refinement probability* for the vulnerability as the probability that some existing refinement task (see Table I) will succeed before the deadline, using the tasks' corresponding performance profiles.
3) Discount the success probability of all adaptation tasks (see Table I) for that vulnerability by the time-bounded refinement probability.

The time-bounded refinement failure probability $P_f(v)$ for a vulnerability v with deadline $d(v)$ and refinement tasks $R(v)$ is computed as follows:

$$P_f(v) = \prod_{k \in R(v)} \begin{cases} 1 - P(k) & \text{if } d(v) = \emptyset \\ 1 & \text{if } t_{now} + t(k) > d(v) \\ 1 - P(k) & \text{otherwise} \end{cases} \tag{4}$$

As a pending deadline approaches for a vulnerability, each existing refinement task for the vulnerability will eventually become impossible to execute before the deadline. This means the time-bounded refinement probability will reach zero via a step function. At this point, the success probability of all adaptation tasks for the vulnerability— which have a high probability of success— will not be discounted, and FUZZBUSTER will satisfy the deadline by sequentially selecting adaptation tasks with high utility.

FUZZBUSTER is not guaranteed to satisfy the deadline by the exact time specified, but it does guarantee that adaptation tasks take precedence at or before that time. If the deadline is set to the present clock time, or to a time in the past, FUZZBUSTER will immediately attempt to adapt the application until the deadline is satisfied.

We next describe experiments that show how the extended AMP-based FUZZBUSTER meta-control provides efficient,

context-sensitive task management that produces the desired types of behavior.

IV. Experiments

We describe five indicative examples from a suite of evaluations that demonstrate FUZZBUSTER's meta-control within the cybersecurity domain. Each experiment illustrates adaptive meta-control under different circumstances, including heavy application usage, balancing multiple types of fuzz-testing, handling vulnerabilities of varying lethality, and handling multiple adaptation deadlines.

A. Balanced fuzz-testing with an application usage schedule

Our first two experiments evaluate FUZZBUSTER's effort-balancing across applications. Since these two experiments focus on evaluating the effort balance of meta-control, we used stub applications in place of real ones, so that FUZZBUSTER would never find and adapt to real vulnerabilities. We present both examples, and then we describe a mathematical evaluation of these behaviors.

In our first example, FUZZBUSTER is given a simple alternating phase schedule, where either application `app1` or `app2` is used in each phase.

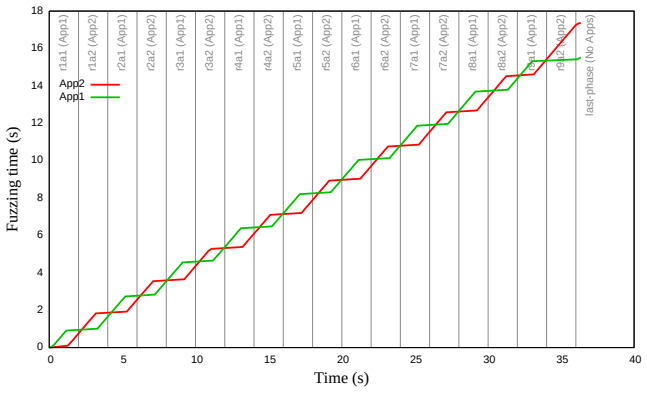

Figure 2. FUZZBUSTER balances the time it spends on two applications based on a schedule of future application usage. This behavior results from the AMP extension for effort balancing, as well as the native AMP tendency to address more imminent threats to help secure future rewards.

Figure 2 illustrates the results. Since FUZZBUSTER represents each application usage as a threat with positive lethality, FUZZBUSTER computes the most utility for tasks that explore applications that will be used in the near future. This means that immediately before an application (*e.g.*, `app1`) is used in a phase (*e.g.*, *r2a1*)— and early-on within that phase— tasks corresponding to the application have relatively high utility, increasing their selection potential. This causes the alternating time allocation behavior of FUZZBUSTER in Figure 2.

In a second example, we designed a sequence of mission phases that simulates a day of application usage by a hypothetical military planner. We repeated the phases twice to simulate a two-day mission. Both days consist of the following phases:

- *Overnight*: no applications are used.
- *Intelligence-gathering* uses the `GIS` and `SensorDashboard` applications.
- *Planning* makes use of `GIS`, `comms`, `PowerPoint`, and `Word`.
- *Briefing* makes use of `PowerPoint`.
- *Exec* makes use of `GIS` and `comms`.
- *Debrief* makes use of `GIS`, `PowerPoint`, and `Word`.

After the first day $d0$ of phases, an identical sequence follows for the next day $d1$.

We used this sequence of phases as input to FUZZBUSTER, and FUZZBUSTER subsequently created and executed discovery tasks for each of these applications.

Results are plotted in Figure 3. Phase boundaries are plotted and labeled along the x-axis, indicating which applications are used in which phase. Each application is plotted as a separate dataset, showing how much time FUZZBUSTER has allocated to fuzzing each application.

As mentioned above, FUZZBUSTER computes the most utility for tasks that explore applications that will be used in the near future. Consequently, since `PowerPoint` is the only application used in the two *briefing* phases, FUZZBUSTER focuses most of its time on addressing that portion of the attack surface prior to (and during) those phases.

The graph also illustrates FUZZBUSTER balancing its fuzz-testing based on future application usage. For instance, in the *overnight* phases, FUZZBUSTER balances its time among multiple applications based on their usage in upcoming phases, and after the final phase of application usage ($d1p5$), FUZZBUSTER attempts to equalize its time allocation across all applications.

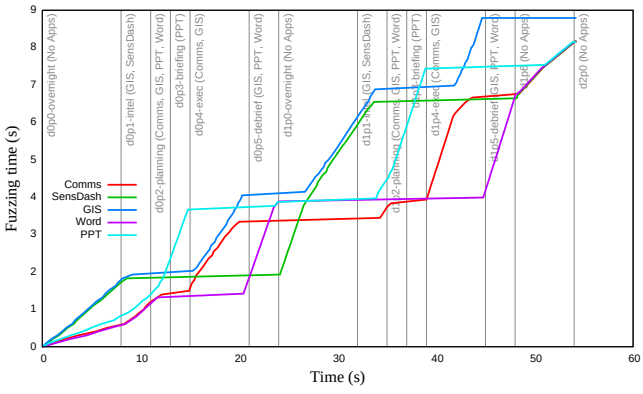

Figure 3. FUZZBUSTER balances the time it spends on applications using a repeating schedule of twelve application usage phases.

We can mathematically evaluate the effort balancing as the time FUZZBUSTER has spent fuzz-testing an application before and during each subsequent usage, as a preparedness measure. This is equivalent to computing the area under

the curve for each given application's dataset, when the application is in use, and then summing over all applications and all phases.

The baseline for comparison is the same area computation if FUZZBUSTER spends equal time on all applications. In the first example in Figure 2, balancing outperforms the baseline by 6%, and in the second example in Figure 3, balancing outperforms the baseline by 9%.

B. Coordinating discovery tasks and refinement tasks

When FUZZBUSTER discovers a vulnerability in its applications, it must quickly refine the vulnerability and then adapt the application. Addressing a known vulnerability should take precedent over additional discovery tasks, since the ultimate goal of FUZZBUSTER is system security.

To test this tradeoff of discovery, refinement, and adaptation tasks (see Table I for the full listing), we provided FUZZBUSTER two faulty applications (`tcsh` and `vuln-ping`). Unlike the previous experiment, we did not provide a phase schedule— those two applications are the only ones of interest.

The results are plotted in Figure 4. FUZZBUSTER begins by creating and executing vulnerability discovery tasks for both applications. It finds a vulnerability in `tcsh`, refines it fully at 25 seconds, and adapts it at 30 seconds. After `tcsh` is adapted, FUZZBUSTER balances its application usage by performing discovery tasks on `vuln-ping`, until around 53 seconds, when it achieves its balance and proceeds to fuzz-test both applications in an alternating pattern.

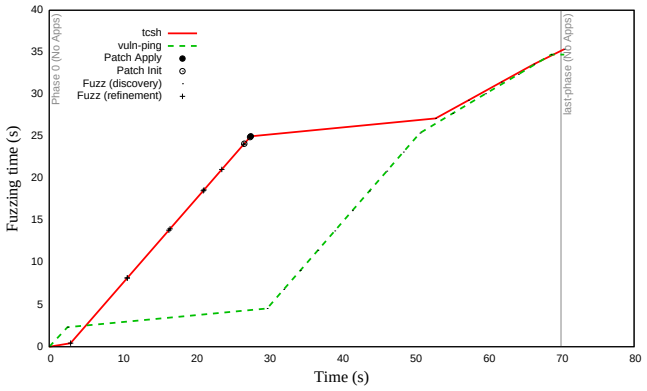

Figure 4. FUZZBUSTER discovers a new vulnerability, immediately addresses it with refinement and adaptation (patch init/apply) tasks, and then resumes its discovery tasks to find novel vulnerabilities and balance its time between applications.

C. Adapting to multiple vulnerabilities of varying lethality

Reactive vulnerabilities that are reported by FUZZBUSTER's underlying host may have resulted from an attack by an adversary, so these pose a higher threat than proactive vulnerabilities discovered by FUZZBUSTER (all else being equal). Consequently, when faced with vulnerabilities of both types, FUZZBUSTER should dedicate more of its effort to adapting to the reactive vulnerabilities.

FUZZBUSTER automatically awards reactive vulnerabilities an order of magnitude higher lethality than proactive vulnerabilities, so tasks that address reactive vulnerabilities increase survival probability— and the likelihood of sustaining future rewards— significantly more than tasks that address proactive vulnerabilities.

In this experiment (plotted in Figure 5), FUZZBUSTER is given a fault-injected desktop calculator application `dc-modfail`, and it begins by executing discovery tasks. It first finds a *proactive* vulnerability `Vuln-dc-modfail-nosrc-1`, plotted in green, and begins executing refinement tasks to better characterize the underlying vulnerability.

At 20 seconds, FUZZBUSTER receives a *reactive* fault notification from the host regarding a vulnerability in the same application (*i.e.*, the fault was not triggered by FUZZBUSTER's own tests of the application). FUZZBUSTER responds by replicating the fault and constructing a vulnerability profile for the reactive vulnerability `Vuln-dc-modfail-nosrc-2`, plotted in blue. FUZZBUSTER stops refining the proactively-discovered vulnerability and proceeds to refine and adapt the application to address the reactive vulnerability, since it poses a greater threat.

After the reactive vulnerability is addressed, FUZZBUSTER continues refining the proactive vulnerability until a second fault notification arrives after 50 seconds. Like the previous reactive vulnerability, FUZZBUSTER subsequently replicates it, records it as `Vuln-dc-modfail-nosrc-3`, refines the vulnerability, and patches it before returning to refining the proactive vulnerability that FUZZBUSTER discovered on its own.

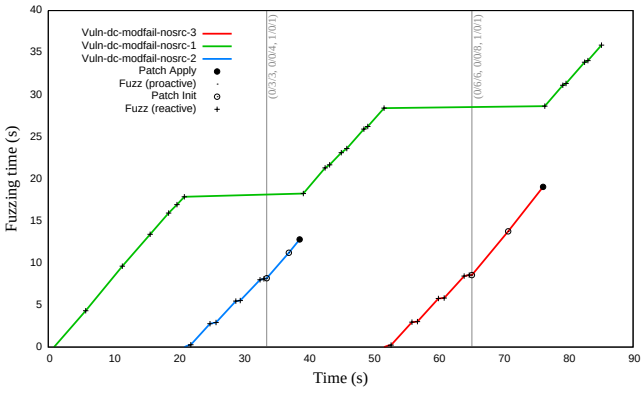

Figure 5. FUZZBUSTER discovers a new proactive vulnerability (green line) and begins refining it, then it receives two unexpected, reactive fault notifications from the host and addresses the reactive vulnerabilities (blue and red lines) before returning to its proactive fuzz-testing.

D. Handling adaptation deadlines

In our final experiment, FUZZBUSTER fuzz-tests three faulty applications (`tcsh`, `guess`, and `vuln-ping`), and when it finds a vulnerability, it creates a 15-second adaptation deadline.

Results are plotted in Figure 6. FUZZBUSTER discovers a vulnerability in `guess` and immediately refines and patches it before the adaptation deadline. After additional proactive fuzz-testing, FUZZBUSTER discovers a vulnerability in `tcsh` and refines it until the adaptation deadline, when the utilities of the adaptation tasks surpass those of the refinement tasks. Once the deadline is satisfied, FUZZBUSTER continues refining the vulnerability and then completes a final adaptation. FUZZBUSTER then proactively fuzzes `vuln-ping` and `guess` to regain its balance across applications.

FUZZBUSTER does not complete its self-adaptation before either deadline— in both cases, it begins its adaptation tasks before the deadline, but finishes adapting roughly 2.5 seconds after either deadline. Since FUZZBUSTER does not spend time planning to coordinate tasks that are not already in existence, it will not work backwards from the deadline to schedule its adaptation; however, it is guaranteed to start adapting before or on the deadline.

The dashed `FB Internals` curve in Figure 6 plots the cumulative time spent on all FUZZBUSTER overheads, including startup, task creation, utility computation, and logging. This overhead comprises less than 3% of FUZZBUSTER's time and, subtracting startup, it comprises less than 1.5% of FUZZBUSTER's time. Thus FUZZBUSTER's meta-control is highly efficient, imposing little overhead for the intelligent self-guiding behavior it provides.

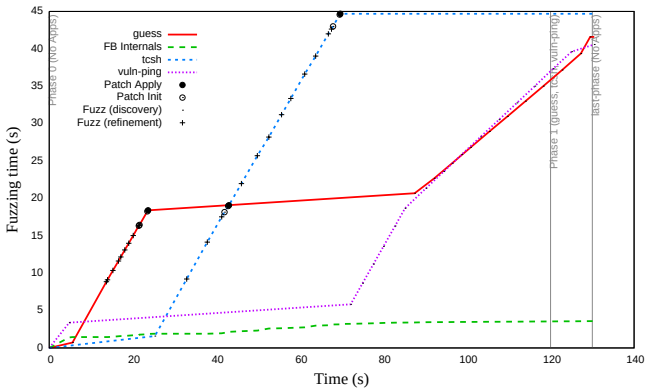

Figure 6. FUZZBUSTER discovers proactive vulnerabilities in two applications, stops refining them by their adaptation deadlines (15s after discovery), and adapts its application. In the case of `tcsh`, FUZZBUSTER performs additional refinement and adaptation after the deadline patch for additional protection.

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 [11]. 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 [12]. 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 [13] and PMOP [14] 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 [15] 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.

Reinforcement learning (RL) is another approach to soft-greedy action selection. Temporal difference RL methods incrementally improve an action policy based on immediate rewards received by an agent, and evolutionary RL methods simulate the process of natural selection to optimize a population of candidate policies. Both of these methods require orders of magnitude more training than our utility-based approach, although these methods can be combined to speed up online learning [16].

VI. 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 orchestrating a diverse set of discovery, refinement, and adaptation tools without incurring a cumbersome overhead for planning and deliberation.

We presented a greedy utility-based task selection algorithm that addresses important factors in the cybersecurity domain. We described two experiments that demonstrate that FUZZBUSTER's meta-control adjusts its task selection to fuzz-test applications based on their usage schedule, giving priority to applications in order of usage and frequency of future usage. When FUZZBUSTER discovers vulnerabilities from fuzz-testing, it immediately refines and repairs the vulnerability, and then returns to discovery tasks, as shown in our third experiment. Our fourth experiment shows that FUZZBUSTER attends to vulnerabilities in order of the threat they pose to the host, so FUZZBUSTER temporarily sets aside less-critical vulnerabilities to repair more threatening ones. Finally, our fifth experiment shows that FUZZBUSTER can deploy temporary adaptations to shield the host while it develops a more accurate final repair. We also showed that FUZZBUSTER's meta-control can attend to these cybersecurity factors and orchestrate adaptations without incurring cumbersome overhead for planning and meta-reasoning.

Next steps for improving FUZZBUSTER's meta-control include using a sliding-window for effort-balancing, updating the lethality of a vulnerability as new refinements are made, and using domain-specific knowledge to more accurately estimate vulnerability lethality and the probability of latent vulnerabilities in applications. These improvements will require little or no extension to the meta-control framework.

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, 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.

[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] D. J. Musliner, R. P. Goldman, and K. D. Krebsbach, "Deliberation scheduling strategies for adaptive mission planning in real-time environments," in *Proc. Third International Workshop on Self Adaptive Software*, 2003. [Online]. Available: http://www.musliner.com/david/papers/safer03.pdf

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] S. Whiteson, M. E. Taylor, and P. Stone, "Empirical studies in action selection with reinforcement learning," *Adaptive Behavior*, vol. 15, no. 1, pp. 33–50, 2007.