Using Concolic Testing to Refine Vulnerability Profiles in FUZZBUSTER

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

***Abstract*—Vulnerabilities in today's computer systems are relentlessly exploited by cyber attackers armed with sophisticated vulnerability search and exploit development toolkits. To protect against such threats, we are developing FUZZBUSTER, an automated system that provides adaptive immunity against a wide variety of cyber threats. FUZZBUSTER uses custom and off-the-shelf fuzz-testing tools to find vulnerabilities, create vulnerability profiles identifying the inputs that drive target programs to the corresponding faults, and synthesize adaptations that prevent future exploits. We have adapted the CREST concolic testing tool so that FUZZBUSTER can refine a vulnerability profile by extracting the symbolic constraints stemming from concrete execution of a target program. This novel use of concolic testing enables FUZZBUSTER to automatically generalize a single fault-inducing input example into a symbolic description of the vulnerability, and thus create more effective adaptations.**

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

I. INTRODUCTION

Modern computer systems face constant attack by sophisticated adversaries, and the number of cyber-intrusions increases every year [1], [2]. Cyber-attackers use numerous vulnerability scanning tools that automatically probe target software systems for a wide array of vulnerabilities. For example, attackers use fuzz-testing tools (such as Peach [3] and SPIKE [4]) that try to crash target applications and SQL injection tools (such as sqlmap [5] and havij [6]) that attempt to manipulate the contents of databases. Upon discovering a potential vulnerability, attackers use powerful exploit development toolkits (such as Metasploit [7] and Inguma [8]) to quickly craft exploits that take advantage of vulnerabilities.

We are developing FUZZBUSTER under DARPA's Clean-slate design of Resilient, Adaptive, Survivable Hosts (CRASH) program to provide self-adaptive immunity from these and other cyber-threats. FUZZBUSTER provides long-term immunity against both observed and novel (zero-day) cyber-attacks. For an in-depth discussion of FUZZBUSTER's capabilities, see [9], [10].

As shown in Figure 1, FUZZBUSTER operates *proactively* to find vulnerabilities before they can be exploited, and *reactively* to address exploits observed "in the wild." FUZZBUSTER directs the execution of custom and off-the-shelf *fuzz-testing* tools to find and characterize vulnerabilities. Fuzz-testing tools find software vulnerabilities by exploring millions of semi-random inputs to a program. FUZZBUSTER creates a *vulnerability profile* representing the nature of each vulnerability, including the ranges of inputs that lead to the vulnerability. These vulnerability profiles represent as much of the vulnerability as FUZZBUSTER can identify. After constructing a vulnerability profile, FUZZBUSTER synthesizes defenses to shield or repair the flaw, protecting against entire classes of exploits that may be encountered in the future.

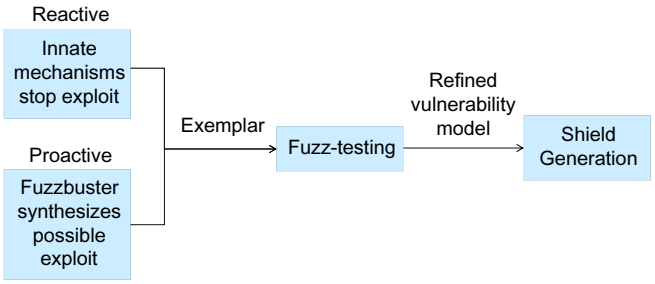

Figure 1. When reacting to a fault, FUZZBUSTER creates an exemplar test case that reflects the environment and inputs at the time of the observed fault. During proactive exploration, FUZZBUSTER synthesizes exemplar test cases that could lead to a fault.

FUZZBUSTER uses semi-random inputs and other fuzz-testing tools to find and replicate vulnerabilities. When a vulnerability is discovered, FUZZBUSTER tries to determine the inputs (command line arguments, inputs to `stdin`, environment variables) relevant to that vulnerability. FUZZBUSTER also uses test tools that try to minimize fault-inducing inputs by eliminating content that is not needed to trigger the fault. FUZZBUSTER then attempts to synthesize an adaptation from the refined vulnerability profile. Unfortunately, adaptations based on information gathered with black-box tools tend to be too narrowly focused, failing to block all fault-inducing inputs. When source code is available, FUZZBUSTER uses concolic testing (a combination of concrete and symbolic execution) to find more general constraints specifying the input values that lead to the fault. Once FUZZBUSTER has eliminated irrelevant parts of the inputs, concolic testing can generalize the vulnerability profile to cover a broader range of fault-inducing inputs.

In this paper, we describe FUZZBUSTER's novel use of the CREST concolic testing tool to identify the inputs that cause a target program to reach a fault.

II. CREST

CREST [11] is an open source concolic testing tool for C programs. As with other concolic testing tools [12], [13], [14], [15], it uses a combination of symbolic and concrete execution to drive a target program through many different code paths.

CREST uses C Intermediate Language (CIL) [16] to instrument a target program to simultaneously perform symbolic and concrete execution. To use CREST normally, a user identifies a set of variables that he is concerned with and CREST rewrites the target program so that those "symbolic" variables can be controlled and tracked. During each test iteration, CREST writes an inputs file, executes the instrumented program and updates its internal state with the results of the run. Each time the instrumented program executes, it initializes the symbolic variables with the values from the inputs file. If there are not enough inputs or if no inputs are specified, the target program initializes each symbolic variable with a random value. As it executes, the target program records its execution, including the loads, stores, assignments, branches, function calls, and returns that relate to the symbolic (traced) variables. User-specified search modes, such as random or depth first, use this execution information to choose inputs for subsequent test iterations. CREST runs test iterations until it executes every code path in the target program or meets other stopping criteria.

III. Approach

Traditionally, concolic test tools try to find a set of inputs that maximize test coverage. In contrast, FUZZBUSTER uses CREST to accumulate the symbolic constraints for a single fault-inducing test case. The resulting constraints specify the range of values the inputs can take without changing the execution path.

We implemented this functionality by adding a new, "concrete" execution mode to CREST that executes the target program once and then prints the constraints for the branches taken during the run. When the instrumented program executes a load instruction on a symbolic variable in this concrete mode, the instrumented code creates a concrete binding and returns the value already stored in the memory location being accessed. Using the already-stored value instead of a random one results in the target program following the desired path through the code. However, this requires additional infrastructure to provide the specified command line arguments and standard inputs.

While refining vulnerability profiles, FUZZBUSTER works with fault-inducing test cases that were either observed in the wild or found by proactive searches. If test programs have suitable source code available, FUZZBUSTER "crestifies" the target program, creating a new version of the source code containing additional statements. The "crestified" version calls functions to identify command line arguments and inputs from `stdin` as symbolic variables. Then, FUZZBUSTER executes CREST on the "crestified" target program using the new concrete mode. Because the symbolic variables capture the command-line arguments and standard inputs to the target program, the resulting constraints specify how an end-user (or attacker) can invoke the target program to induce a fault. This same symbolification can be performed for environment variables, file inputs and network streams.

IV. Example

Listing 1. `100over.c`

```
1 /* Divides 100 by a number.
2  *
3  * If the divisor is zero a Floating
4  * Point Exception will result
5  */
6 #include <stdio.h>
7
8 int main (int argc, char *argv[]) {
9   int d = 0;
10  int r = 0;
11
12  d = 1;
13  if (argc >= 2) {
14    d = atoi(argv[1]);
15  }
16  r = 100 / d;
17  printf("%d\n", r);
18  return 0;
19 }
```

To illustrate FUZZBUSTER's process in more detail, we now describe a simple example based on a trivial, deliberately-faulty program. Listing 1 contains the source code for `100over`, our simple brittle program that converts its first argument to an integer and divides it into 100. If the argument is 0, `100over` crashes with a floating point error. Suppose that FUZZBUSTER detected this fault, either by random proactive fuzz-testing or by observing a user triggering the fault accidentally. FUZZBUSTER would begin by trying to refine a new vulnerability profile to describe the fault. This might include trying to determine which inputs to the program are critical to causing the fault; in particular, testing to see if any environment variables, command-line arguments, or `stdin` inputs affect the reproducibility of the fault. Having determined that the command-line arguments are important and because the `100over` source code is available, FUZZBUSTER could then use CREST to derive a more precise description of what kinds of command line arguments cause the fault.

Listing 2. Segment from `100over_crest.c`

```
1 int main (int argc, char *argv[]) {
2   /* CRESTify argc */
3   CREST_int(argc);
4   /* CRESTify argv */
5   int _i; // argc index
6   int _j; // argv[_i] index
7   int _len; // strlen(argv[_i])
8   for (_i = 1; _i < argc; ++_i) {
9     _len = strlen(argv[_i]);
10    // save null also
11    for (_j = 0; _j <= _len; ++_j) {
12      CREST_char(argv[_i][_j]);
13    }
14  }
15  /* CRESTify sentinel before main
16   * variables */
17  int _crestify = -889270259;
18  CREST_int(_crestify);
19  /* main variables */
20  int d = 0;
21  int r = 0;
22  /* CRESTify main variables */
23  CREST_int(d);
24  CREST_int(r);
25  /* install CRESTify signal handler
        */
26  int _signums[] = {SIGHUP, SIGINT,
27                    SIGQUIT, SIGILL,
28                    SIGABRT, SIGFPE,
29                    SIGUSR1, SIGSEGV,
30                    SIGUSR2, SIGALRM,
31                    SIGTERM, SIGSYS,
32                    0};
33  for(_i = 0; _signums[_i] > 0; ++_i)
34   {signal(_signums[_i], gotsignal);}
35  /* original program follows */
36  d = 1;
37  if (argc >= 2) {
38    d = atoi(argv[1]);
39  }
40  r = 100 / d;
41  printf("%d\n", r);
42  return 0;
43 }
44 void gotsignal(int signum)
45 __attribute__((__crest_skip__))
46 {
47  psignal(signum, "got signal");
48  __CrestSignalHandler();
49  exit(1);
50 }
```

FUZZBUSTER's "crestify" script transforms the `100over` `main` function into the form shown in Listing 2. The crestified code calls `CREST_int` and `CREST_char` to identify user-controlled variables as symbolic. When CREST runs the crestified program, all accesses (loads, stores, assignments) of these variables are recorded. The crestified code also sets up a signal handler so that if a fault is encountered (such as the floating point error), the target program will exit cleanly and the execution information will be available to CREST.

Listing 3. Segment from `100over_crest.cil.c`

```
1   __CrestLoad(297, (unsigned long )0,
        (long long )1);
2   __CrestStore(298, (unsigned long )(&
        d));
3 #line 156
4   d = 1;
5   {
6   __CrestLoad(301, (unsigned long )(&
        argc), (long long )argc);
7   __CrestLoad(300, (unsigned long )0,
        (long long )2);
8   __CrestApply2(299, 17, (long long )(
        argc >= 2));
9 #line 157
10  if (argc >= 2) {
11    __CrestBranch(302, 105, 1);
12 #line 158
13    mem_12 = argv + 1;
14 #line 158
15    d = atoi((char const    *)*mem_12);
16    __CrestHandleReturn(305, (long
        long )d);
17    __CrestStore(304, (unsigned long )
        (& d));
18  } else {
19    __CrestBranch(303, 106, 0);
20
21  }
22  }
```

When CREST runs on the target program, it uses CIL to add additional instrumentation (see the code segment in Listing 3). CREST links this instrumented file with a library that records the operations. CREST can only track memory accesses and branches in instrumented source code, so the target program either needs to include the implementations of all functions (such as `atoi` and `strlen`) or needs to be linked with libraries that were instrumented by CREST.

As shown in Figure 2, running CREST on `100over` with "0" as the command line argument yields five initial values and eleven constraints. The constraints describe the bounds on input variables that will always lead to the same faulty code, *i.e.*, the vulnerability.

```
initial 0: argc = 2; # int
initial 1: argv[1][0] = '0'; # char
initial 2: argv[1][1] = '\0'; # char
initial 3: _crestify = -889270259; # int
initial 4: d = 0; # int
initial 5: r = 0; # int
constraint 0: (< (+ 1 (* -1 argc)) 0)
constraint 1: (>= (+ 2 (* -1 argc)) 0)
constraint 2: (>= (+ -2 (* 1 argc)) 0)
constraint 3: (/= (+ -32
                     (* 1 argv[1][0])) 0)
constraint 4: (/= (+ -45
                     (* 1 argv[1][0])) 0)
constraint 5: (/= (+ -43
                     (* 1 argv[1][0])) 0)
constraint 6: (>= (+ -48
                     (* 1 argv[1][0])) 0)
constraint 7: (<= (+ -57
                     (* 1 argv[1][0])) 0)
constraint 8: (< (+ -58
                    (* 1 argv[1][0])) 0)
constraint 9: (< (+ -48
                    (* 1 argv[1][1])) 0)
constraint 10: (= (+ -48
                     (* 1 argv[1][0])) 0)
```

Figure 2. Running CREST with the concrete search mode yields two lists: the initial variable assignments and the constraints corresponding to the branches taken during concrete execution.

In Figure 2, constraints 0 and 1 come from the nested loops at the start of `main` (lines 8–14 in Listing 2) that make the contents of `argv` symbolic. Constraint 2 reflects the check to ensure that `argv[1]` is set (line 39 in Listing 2). Constraints 3 through 9 reflect the branches in the source code for `atoi`. Constraint 3 states that the character is not a space (ASCII code 32). Constraints 4 and 5 state that the character is not a "-" or "+" (ASCII codes 45 and 43). Constraints 6 and 7 enforce that the character is a digit. Constraints 8 and 9 reflect the end conditions for the main loop in `atoi`.

The final constraint, 10, reflects the condition for the floating point exception— *i.e.*, that the input is exactly zero. This last constraint is added by the `__CrestSignalHandler()` that we developed (see line 51 in Listing 2). When the target program throws the floating point exception, this signal handler records all of the symbolic expressions on the stack, adding constraints specifying that these expressions must equal the concrete value stored in memory. Without this special signal-handling code, CREST cannot tell that a branch occurred and cannot record a suitable constraint.

V. Next Steps

FUZZBUSTER currently parses the constraints identified by CREST and updates the vulnerability profile accordingly. We are working on several extensions that will enable FUZZBUSTER to synthesize security-improving adaptations directly from the identified constraints.

FUZZBUSTER currently generates adaptations that wrap the target program and remove fault-inducing inputs or prevent execution when they are present. In the future, FUZZBUSTER could generate an adaptation that checks if inputs match every identified constraint and, if so, aborts the execution of the target program. Such an adaptation would have minimal impact on normal operation, but might not catch other execution paths that lead to the fault. Alternatively, FUZZBUSTER could generate an adaptation that checks an individual constraint and aborts execution if that constraint is met. This type of adaptation would have potentially broad coverage, but might have a negative impact on normal operation. Additionally, FUZZBUSTER could generate adaptations that check any subset of the identified constraints.

Because FUZZBUSTER regression tests adaptations before applying them, it could generate numerous adaptations from the constraints and then test them to find an adaptation with a suitable balance between fault prevention and impact on normal operation. Thus, FUZZBUSTER could take into account the precise conditions leading to a fault and prevent the fault with limited impact to the normal (non-faulting) execution of the target program.

For many target programs, CREST may generate sets of constraints where not every constraint must be satisfied to reach the fault. For instance, a command-line argument triggering verbose output could result in numerous constraints, but might not influence the overall execution of the target program. We could identify such constraints by constructing inputs that violate one of the constraints at a time to see if that constraint is required to reach the fault.

More generally, a target program may contain numerous paths that lead to the faulting location; FUZZBUSTER also needs to protect against inputs driving the target program down these paths. To identify these paths, we could add another mode to CREST that uses the control flow graph to search for additional paths that reach the fault's location.

In addition to creating wrapper adaptations that protect vulnerable programs by modifying inputs, FUZZBUSTER uses the evolutionary program repair tool, GenProg [17] to synthesize source code adaptations. GenProg uses a regression test suite to identify potential sites for source code modifications. FUZZBUSTER could use the constraints identified by CREST to provide guidance to GenProg about what source code sites are relevant. This would speed GenProg's operation and increase the chances of finding a suitable repair.

VI. Related Work

As previously noted, the FUZZBUSTER approach has roots in fuzz-testing, a term first coined in 1988 applied to software security analysis [18]. 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 [19]. 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 [20] and PMOP [21] 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 [22] 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.

VII. Conclusion and Future Work

FUZZBUSTER is intended to augment and eventually outmode various post-exploit security tools such as virus scanners. Rather than scanning a computer all night to see if it has been compromised by an exploit, FUZZBUSTER will scan for vulnerable software and repair or shield it. FUZZBUSTER's novel use of CREST is a major step toward the fully automatic use of concolic testing for vulnerability refinement and protection.

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] "Peach fuzzing platform." [Online]. Available: http://peachfuzzer.com/

[4] Immunity, "Spike." [Online]. Available: http://www.immunitysec.com/resources-freesoftware.shtml

[5] "Sqlmap." [Online]. Available: http://sqlmap.sourceforge.net

[6] "Havij." [Online]. Available: http://www.itsecteam.com/en/projects/project1.htm

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

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

[9] 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.

[10] ——, "Fuzzbuster: A system for self-adaptive immunity from cyber threats," in *Eighth International Conference on Autonomic and Autonomous Systems (ICAS-12)*, March 2012.

[11] 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

[12] P. Godefroid, M. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[13] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[14] 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.

[15] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2005, pp. 263–272.

[16] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Horspool, Ed. Springer Berlin / Heidelberg, 2002, vol. 2304, pp. 209–265.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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