FuzzBomb: Autonomous Cyber Vulnerability Detection and Repair

David J. Musliner, Scott E. Friedman, Michael Boldt, J. Benton, Max Schuchard, Peter Keller
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {dmusliner,sfriedman,mboldt}@sift.net
Stephen McCamant
University of Minnesota
Minneapolis, USA
email: mccamant@cs.umn.edu

***Abstract*—Beginning just over one year ago, Smart Information Flow Technologies (SIFT) and the University of Minnesota teamed up to create a fully autonomous Cyber Reasoning System (CRS) to compete in the Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge (CGC). Starting from our prior work on autonomous cyber defense and symbolic analysis of binary programs, we developed numerous new components to create FuzzBomb. In this paper, we outline several of the major advances we developed for FuzzBomb, and review its performance in the first phase of the CGC competition.**

***Keywords*-autonomous cyber defense; symbolic analysis; protocol learning; binary rewriting.**

I. Introduction

In June 2014, DARPA funded seven teams to build autonomous CRSs to compete in the DARPA CGC. SIFT and the University of Minnesota (UMN) together formed the FuzzBomb team, building on our prior work on the Fuzzbuster cyber defense system [1] and the FuzzBALL symbolic analysis tool [2].

SIFT's Fuzzbuster system automatically finds flaws in software using symbolic analysis tools and fuzz testing, refines its understanding of the flaws using additional testing, and then synthesizes *adaptations* (e.g., input filters or source-code patches) to prevent future exploitation of those flaws, while also preserving functionality. Fuzzbuster includes an extensible plug-in architecture for adding new analysis and adaptation tools, along with a time-aware, utility-based meta-control system that chooses which tools are used on which applications during a mission [3]. Before the CGC began, Fuzzbuster had already automatically found and shielded or repaired dozens of flaws in widely-used software including Linux tools, web browsers, and web servers.

In separate research, Prof. Stephen McCamant at UMN had been developing the FuzzBALL tool to perform symbolic analysis of binary x86 code. FuzzBALL combines static analysis and symbolic execution to find flaws and proofs of vulnerability through heuristic-directed search and constraint solving. On a standard suite of buffer overflow vulnerabilities, FuzzBALL found inputs triggering all but one, many with less than five seconds of search [2].

Together, Fuzzbuster and FuzzBALL provided the seeds of a strategic reasoning framework and deep binary analysis methods needed for our FuzzBomb CRS. However, many challenges still had to be addressed to form a fully functioning and competitive CRS. In this paper, Section II describes the CGC competition, Sections III and IV overview our prior components, Section V outlines several of the major advances we developed for FuzzBomb, and Section VI reviews its performance in the first phase of the CGC competition.

II. DARPA's Cyber Grand Challenge

Briefly, the CGC is designed to be a simplified form of Capture the Flag game in which DARPA supplies Challenge Binaries (CBs) that nominally perform some server-like function, responding to client connections and engaging in some behavioral protocol as the client and server communicate. The CBs are run on a modified Linux operating system called Decree, which provides a limited set of system calls. In the competition, CBs are provided as binaries only (no source code) and are undocumented, so the CRSs have no idea what function they are supposed to perform. However, in some cases a network packet capture (PCAP) file is provided, giving noisy, incomplete traces of normal non-faulting client/server interactions ("pollers"). Within each CB is one or more vulnerability that can be accessed by the client sending some inputs, leading to a program crash. To win the game, a CRS must find the vulnerability-triggering inputs (called Proofs of Vulnerability (PoVs)) and also repair the binary so that the PoVs no longer cause a crash, and all non-PoV poller behavior is preserved. The complex scoring system rewards finding PoVs, repairing PoVs, and preserving poller behavior, and penalizes increases in CB size and decreases in CB speed.

III. Background: Fuzzbuster

Since 2010, we have been developing Fuzzbuster under DARPA's Clean-Slate Design of Resilient, Adaptive, Secure

ISBN: 978-1-61208-444-2

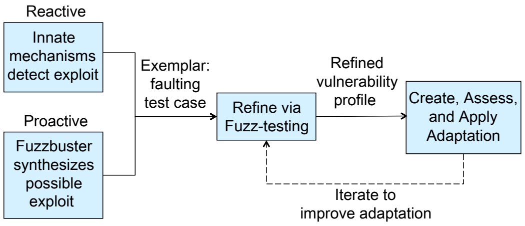
Reactive
Innate mechanisms detect exploit
Proactive
Fuzzbuster synthesizes possible exploit
Exemplar: faulting test case
Refine via Fuzz-testing
Refined vulnerability profile
Create, Assess, and Apply Adaptation
Iterate to improve adaptation

Figure 1. Fuzzbuster refines both proactive and reactive fault exemplars into vulnerability profiles, then develops and deploys adaptations that remove vulnerabilities.

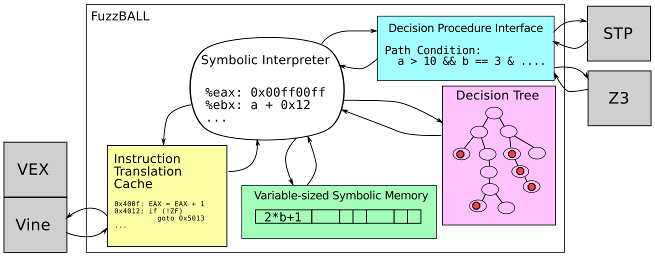

Figure 2. An overview of our FuzzBALL binary symbolic execution engine.

Hosts (CRASH) program to use software analysis and adaptation to defeat a wide variety of cyber-threats. By coordinating the operation of automatic tools for software analysis, test generation, vulnerability refinement, and adaptation generation, Fuzzbuster provides long-term immunity against both observed attacks and novel (zero-day) cyber-attacks.

Fuzzbuster operates both *reactively* and *proactively*, as illustrated in Figure 1. When an attacker deploys an exploit and triggers a program fault (or other detected misbehavior), Fuzzbuster captures the operating environment and recent program inputs into a *reactive exemplar*. Similarly, when Fuzzbuster's own software analysis and fuzz-testing tools proactively create a potential exploit, it is summarized in a *proactive exemplar*. These exemplars are essentially tests that indicate a (possible) vulnerability in the software, which Fuzzbuster must characterize and then shield from future exploitation. For example, an exemplar could hold a particular long input string that arrived immediately before an observed program fault.

Starting from an exemplar, Fuzzbuster uses its program analysis tools and fuzz-testing tools to refine its understanding of the vulnerability, building a *vulnerability profile* (VP). For example, Fuzzbuster can use concolic testing to find that the long-string reactive exemplar is triggering a buffer overflow, and the VP would capture this information. Or, Fuzzbuster can use delta-debugging and other fuzzing tools to determine the minimal portion of the string that triggers the fault.

At the same time, Fuzzbuster tries to create software adaptations that shield or repair the underlying vulnerability. In the simplest case, Fuzzbuster may choose to create a filter rule that blocks some or all of the exemplar input (i.e., stopping the same or similar attacks from working a second time). This may not shield the full extent of the vulnerability (or may be too broad, compromising normal operation), so Fuzzbuster will keep working to refine the VP and develop more effective adaptations. Even symbolic analysis may not yield a minimal description of the inputs that can trigger a vulnerability: there may be many vulnerable paths, only some of which are summarized by a constraint description. Over time, as Fuzzbuster refines the VP and gains a better understanding of the flaw, it may create more sophisticated and effective adaptations, such as filters that block strings based on length not exact content, or actual software patches that repair the buffer overflow flaw.

While Fuzzbuster already had the coordination infrastructure and representation/reasoning to manage exemplars, VPs, and adaptations, many of the tools we had integrated could not apply to the CGC because they do not operate directly on binaries. To fill these gaps and support the full spectrum of vulnerability detection, exploitation, and repair needed for CGC, we integrated with UMN's FuzzBALL and also developed new components, as described below.

IV. Background: FuzzBALL

FuzzBALL is a flexible engine for symbolic execution and automatic program analysis, targeted specifically at binary software. In the following paragraphs we briefly describe the concepts of symbolic execution and explain FuzzBALL's architecture, emphasizing its features aimed at binary code.

The basic principle of symbolic execution is to replace certain *concrete* values in a program's state with *symbolic variables*. Typically, symbolic variables are used to represent the inputs to a program or sub-function, and the symbolic analysis results in an understanding of what inputs can lead to different parts of a program. An interpreter executes the program, accumulating symbolic expressions for the results of computations that involve symbolic variables, and constraints (in terms of those symbols) that describe which conditional branches will occur. These symbolic expressions are valuable because they can summarize the effect of many potential concrete executions (i.e., many possible inputs). When a symbolic expression is used in a control-flow instruction, we call the formula that controls the target a *branch condition*. On a complete program run, the conjunction of the conditions for all the symbolic branches is the *path condition*. We can use an Satisfiability Modulo Theories (SMT) solver (such as Z3 [4]) on a path condition to find a set of concrete input values that would cause the corresponding path to be executed, or to determine what other paths might be feasible.

Many symbolic execution tools operate on program source code (e.g., KLEE, Crest), but FuzzBALL is differentiated by its focus on symbolic execution of binary code. At its core, FuzzBALL is an interpreter for machine (e.g., x86) instructions, but one in which the values in registers and memory can be symbolic expressions rather than just concrete bit patterns. Figure 2 shows a graphical overview of FuzzBALL's architecture. As it explores possible executions of a binary, FuzzBALL builds a *decision tree* data structure. The decision tree is a binary tree in which each node represents the occurrence of a symbolic branch on a particular execution path, and a node has children labeled "false" and "true" representing the next symbolic branch that will occur in either case. FuzzBALL uses the decision tree to ensure that each path it explores is different, and that exploration stops if no further paths are possible.

We have used FuzzBALL on several CGC-relevant research projects, which typically build on the basic FuzzBALL engine by adding heuristics or other features specialized for a particular problem domain. For FuzzBomb and the CGC, we integrated FuzzBALL with the Fuzzbuster reasoning framework and significantly extended FuzzBALL's program analysis capabilities.

V. New Developments

A. Hierarchical Architecture

We designed FuzzBomb to operate on our in-house cluster of up to 20 Dell Poweredge C6100 blade chassis, each holding eight Intel XEON Harpertown quad-core CPUs. To allocate this rack of computers, we designed a hierarchical command-and-control scheme in which different FuzzBomb agents play different roles. At the top of the hierarchy, several agents are designated as "Optimus", or leader agents. At any time, one is the primary leader, known as Optimus Prime (OP). All of the other Optimi are "hot backups," in case OP goes down for any reason (hardware failure, software crash, network isolation). All messages sent to OP are also sent to all of the other Optimi, so that their knowledge is kept up to date at all times. We enhanced our existing fault detection and leader election protocol methods to ensure that an OP is active in the cluster with very high reliability. We usually configure FuzzBomb with three Optimi, each run on a different hardware chassis in the cluster.

Below OP, a set of "FuzzBomb-Master" agents are designated, each to manage the reasoning about a single CB. OP's main job is allocating CBs to those Master agents and giving them each additional resources (other FuzzBombs, DVMs) to use to improve their score on a CB. A FuzzBomb-Master's job is improve its score on its designated CB, using its allocated computing resources in the best way possible (whether that is analysis, rewriting, or testing/scoring). As progress is made on each CB, the responsible FuzzBomb-Master will report that progress and the best-revised-CB-so-far back to OP.

OP's objective is to maximize the system's overall score, keeping in mind deadlines and other considerations. By design, OP should dynamically re-allocate the reasoning assets to the most challenging problems, to maximize the overall system's score. OP is also responsible for uploading FuzzBomb's final best answers to the competition.

B. FuzzBALL Improvements

FuzzBomb uses an improved FuzzBALL symbolic execution engine in an approach that combines ideas from symbolic execution and static analysis in order to find vulnerabilities in binary programs. A static-style analysis identifies parts of the program that might contain a vulnerability. Then a symbolic execution search seeks an execution path from the start of the program to the possible vulnerability point that constitutes a proof of vulnerability. Symbolic execution generates a number of input constraint sets, each set representing a family of related program execution paths. The symbolic execution engine uses these constraint sets to determine the inputs to the program that can reach the program vulnerability, offering a proof-of-concept exploit. While exploring this space, the symbolic execution engine will encounter many decision points (such as conditional branches). Each of these decision points branches off a new set of paths, leading to an exponentially growing number of paths. Exploring this search space of paths represents a significant computational effort. Scaling up the search in a way that mitigates this path explosion poses a key challenge. To overcome this problem, we applied parallelization techniques and heuristic search improvements, as well as other algorithmic changes.

1) Heuristic Guidance: Because the space of program executions is vast, even in the constraint-based representations of symbolic reasoning, heuristic guidance is essential. For the CGC, the key objective is to guide the search towards potential vulnerabilities. FuzzBomb identifies potentially vulnerable instruction sequences and uses abstraction heuristics to focus the search towards those targets. Although a wide variety of source-level coding mistakes can leave a program vulnerable, these dangerous constructs are more uniform when viewed in terms of the binary-level capability they give to an attacker. For example, many types of source-code vulnerabilities create binary code in which the destination of an indirect jump instruction can be influenced by an attacker. The source-code and compiler details about why such a controllable jump arises are often irrelevant, and are not our focus. In particular, FuzzBomb does not try to decompile a binary back to a source language, nor will it identify which particular source code flaw describes a vulnerability. FuzzBomb's search guidance strategies target just these end-result capabilities, e.g., searching for an indirect jump that can be controlled to lead to attack code.

FuzzBomb uses ***problem relaxation heuristics*** to reduce the search space of possible executions, drawing on recent advances in heuristic search techniques for directed symbolic execution and Artificial Intelligence (AI) planning.

ISBN: 978-1-61208-444-2

To search through very large spaces, these techniques use rapid solutions to relaxed or approximate versions of their real problems to provide heuristic guidance. Over the last dozen years, research on relaxation heuristics has produced immense improvements in the scalability of AI planning and other techniques. For example, Edelkamp *et al.* [5] report up to four orders of magnitude reduction in nodes searched in model-checking. Similarly, AI planning systems have gone from producing plans with no more than 15 steps to plans with hundreds of steps (representing many orders of magnitude improvement in space searched). These techniques are only now being applied to directed symbolic execution to help find program paths to vulnerabilities (e.g., Ma *et al.* [6]).

For FUZZBOMB, the problem is to find a symbolic execution path through a program that leads to a vulnerability. One key research challenge is finding the best relaxation method for symbolic execution domains. We developed an approach using causal graph heuristics found in AI planning search [7] to direct symbolic execution, in a manner similar to call-chain backwards symbolic execution [6]. These heuristics use factorization to generate a causal model of subproblems, then "abstract away" interactions between the subproblems to create a relaxed version of the problem that can be solved quickly at each decision point during search. In symbolic execution, solving the relaxed problem determines:

- A reachability analysis to a vulnerability. If the relaxation of the program indicates a vulnerability is unreachable from a particular program decision point, then exploring from that point is fruitless.
- A distance estimate at each decision point that lets exploration proceed along an estimated shortest path.

To generate the relaxation heuristic, FUZZBOMB uses the causal model present within data-flow and control-flow graph (CFG) structures used in binary program analysis. For instance, in a CFG, nodes represent blocks of code and edges represent execution order. This provides a subproblem structure, allowing for bottom-up solving of each subproblem.

2) Other Improvements: The FuzzBALL approach to hybrid symbolic execution and static analysis needed many other improvements to work on the CGC CBs. Our major developments have included:

- Porting to Decree— We adapted FuzzBALL to handle the unique CB format, including emulating the restricted Decree system calls and handling the specific limitations of the CB binary format.
- Improving over-approximated CFG methods— Prior to symbolic analysis, FuzzBALL requires the control flow graph (CFG) of the target binary. Various existing methods are all imperfect at recovering CFGs, but some can be combined. We developed a new CFG-recovery tool that leverages prior work on recursive disassembly along with an updated over-approximation method that finds all of the bit sequences in a binary representing valid addresses/offsets within the binary and treats those as possible jump targets. While this overapproximation is extreme, FUZZBOMB uses heuristics to reduce the size of the resulting CFGs.
- Detecting input-controllable jumps— As FuzzBALL extends branch conditions forward through the possible program executions, whenever it reaches a jump it formulates an SMT query asking whether the CB inputs could force the jump to 42 (i.e., an arbitrary address). If so, a likely vulnerability has been identified.
- Detecting null pointer dereferences, return address overwrites, and various other vulnerable behaviors.
- Making incremental solver calls— We have enhanced FuzzBALL's SMT solver interface so that it can behave incrementally. For example, after querying if a jump target is input-controllable, it can retract that final part of the SMT query and the SMT solver can retain some information it derived during the prior solver call. Microsoft's Z3 SMT solver is state of the art and supports this type of incremental behavior.
- Handling SSE floating point (FP)— The original FuzzBALL implementation used a slow, emulation-based method to handle floating point calculations, and it could not handle the modern SSE FP instructions. We have recently completed major extensions that allow FuzzBALL to handle SSE FP instructions using Z3. We have switched over to using Z3 by default, and are collaborating with both the Z3 and MathSAT5 developers to fix bugs in their solvers and improve their performance.
- Implementing veritesting— David Brumley's group coined this term for a flexible combination of dynamic symbolic execution (DSE) and static symbolic execution (SSE) used to reason in bulk about blocks of code that do not need DSE [8]. We completed our own first version of this capability, along with associated test cases and SMT heuristic improvements. However, as noted below, this improvement was not used during the actual competition because its testing and validation was not complete.

Symbolic execution can be expensive because it is completely precise; this precision ensures that the approach can always create proofs of vulnerabilities. At the same time, it is valuable to know about potentially dangerous constructs even before we can prove they are exploitable. To that end, we modified FuzzBALL to run as a hybrid of static analysis and symbolic execution techniques.

C. Proofs of Vulnerability (PoVs)

We developed two ways of creating PoVs. First, when FuzzBALL identifies a vulnerability that can be triggered by client inputs, it will have solved a set of constraints on the symbolic input variables that describe a class of

PoVs for that vulnerability. Depending on the constraints, the PoV description may be more or less abstract (i.e., it may require very concrete inputs or describe a broad space of inputs that will trigger the vulnerability). For the concrete case, FUZZBOMB has a mechanism to translate FuzzBALL's constraints into the XML format required for a PoV.

Second, if a CB is provided with a PCAP file that illustrates how it interacts with one or more pollers, FUZZBOMB uses protocol reverse engineering techniques to derive an abstract description of the acceptable protocols for a CB. FUZZBOMB then feeds this protocol description into one or more fuzzing tools, to try to develop input XML files that trigger an unknown vulnerability.

We initially developed a protocol reverse engineering tool building on Antunes' ideas [9]. However, the techniques did not scale well to the large numbers of pollers present in the CGC example problems, and they are not robust to the packet loss present in the provided packet captures. We then developed a less elaborate protocol analysis tool which, while not providing a full view of the protocol state machine, allows FUZZBOMB to generate protocol sessions which are accepted by the CBs. This tool uses a heuristic approach, based on observations from prior work in the field [10], to identify likely protocol command elements, fields required for data delivery to the CBs (e.g. message lengths and field offsets), and message delimiters. Additionally, the protocol inference tool also attempts to identify session cookies and simple challenge/response exchanges that are required by the protocol. Significant effort was also required to process the DARPA-provided PCAP files because they contain unexpected packet losses and non-TCP-compliant behavior.

D. Binary Rewriting

We have developed a powerful binary rewriting tool suite that includes mechanism for rewriting instructions, relocating code, and inserting arbitrary code into binaries (if necessary, via trampolining) [11]. Building off of the conservative, over-approximated CFG, these rewriting tools can be used to perform a variety of proactive defensive rewrites as well as focused repairs. For example, FUZZBOMB can inject well-known techniques such as stack canaries that can protect against stack smashing and code injection attacks. We have developed a search-based method for trying different stack canary injection locations, trying to find a location for the canary and the canary-check that preserves known good functionality and defeats known PoVs.

We have also developed an "unstripping" tool that finds the unique bit patterns of the `libcgc` library calls in a binary. FUZZBOMB will use this information to identify which library calls are not used in a particular CB, and their space can be reclaimed for use by the binary rewriting tool (without expanding the size of the binary at all).

Initially, finding space to inject canaries and trampolining code was a major challenge. FUZZBOMB has three methods for finding space to use for rewriting:

- Hijacking program header segments.
- Using file space in between segments (which start on page boundaries).
- Extending a file's executable segment up to the next page boundary.

We have also developed purely defensive rewriting methods that can protect against flaws that FUZZBOMB has not yet identified. Currently, our defensive measures use heuristics to identify functions that receive input data and seem likely to contain a potential overflow flaw. The system then incrementally adds canary-based stack protection to those blocks, re-testing the resulting CB version to see if it still performs as expected. However, the performance (speed) impact of these changes is difficult to assess without many test cases. By default, FUZZBOMB chooses to add stack protection to just three target blocks, chosen heuristically.

VI. RESULTS AND CONCLUSIONS

The first year of CGC involved three opportunities to assess FUZZBOMB's performance: two practice Scored Events (SE1 and SE2) and the CGC Qualifying Event (CQE), which determined which competitors would continue to the second year of competition. In SE1, DARPA released fifteen challenge binaries, some of which had multiple vulnerabilities. At the time, FUZZBOMB had only recently become operational on our computing cluster, and it did not solve many of the problems. However, with access to the source of the SE1 examples and many bug fixes, some months later we had improved FUZZBOMB enough that it was able to find vulnerabilities in four of the problems, including at least one undocumented flaw. For each of those vulnerabilities, FUZZBOMB had a repair that was able to stop the vulnerability from being attacked while also preserving all of the functionality tested by up to 1000 provided test cases. FUZZBOMB also create defensive rewrites for all of the other binaries. In SE2, DARPA provided nine new challenge binaries in addition to the prior fifteen, giving a total of twenty-four. Each problem was supplied with either no PCAPs or a PCAP file containing up to 1000 client/server interactions. At the time of SE2, FUZZBOMB was only able to find two of the new vulnerabilities, but that performance was enough to earn fourth place, when the SE1 problems were included in the ranking.

Our progress in improving the system was slowed by major problems with the government-provided testing system: running parallel tests interfered with each other, and running batches of serialized tests could cause false negatives, hiding vulnerabilities. This meant we had to run tests one at a time, incurring major overhead and making test-running a major bottleneck (especially when given 1000 tests from PCAPs, or when FUZZBOMB created many tests itself). We finally resolved these issues by discarding the provided testing tool and writing our own. Our tool supported safe

ISBN: 978-1-61208-444-2

parallel testing and increased testing speeds by at least two orders of magnitude. However, it took many weeks to come to that conclusion. Several key analysis functions were not completed, including handling challenge problems that had multiple communicating binary programs, complete support for SSE floating point instructions, and veritesting. We also were not able to build the ability to have the system reallocate compute nodes to different CBs or to different functions (DVM vs. running FuzzBALL). By the time of the CQE, in June 2015, FuzzBomb was only able to fully solve seven of the twenty-four SE2 problems. If given the PoVs for the twenty-four problems, the repair system was able to fix twelve CBs perfectly, and the defense system earned additional points on the remaining CBs.

For CQE, DARPA provided 131 all-new problems to the twenty-eight teams who participated (out of 104 originally registered). Each problem was supplied with either no PCAPs or a single client/server interaction. Unfortunately, this singleton PCAP triggered an unanticipated corner case in FuzzBomb's logic: the protocol analysis concluded that every element of the single client/server interaction was a constant, so the protocol had no variables to fuzz. And the default fuzz-testing patterns were not used because there *was* a protocol. Thus FuzzBomb's fuzzing was completely disabled for those challenge problems. Also, because the reallocation functionality was not available, we had to preallocate the number of DVMs vs. FuzzBALL symbolic search engines. We chose to use 325 DVMs and only 156 FuzzBombs, because testing had been such a bottleneck. However, since there were almost no test cases provided in the PCAP files and fuzzing was disabled, FuzzBomb had very few tests to run, and the DVMs were largely idle. With most CBs having only a single FuzzBALL search engine, there was little parallel search activity, and FuzzBomb only found vulnerabilities in 12 CBs (some using prior SE2 PoVs). Of those, with the limited testing available, repair was only able to perfectly fix six (as far as our system could tell). Defense rewrote all of the remaining problems.

When the final CQE scores were revealed, FuzzBomb came in tenth place and did not qualify to continue in the competition (only the top seven teams qualified). In addition to the singleton PCAP files and other issues, we learned of another "curveball" when the scores were released: among the 131 test cases, there were 590 known vulnerabilities, an average of more than 4.5 flaws per binary. In hindsight, FuzzBomb's defensive system should have been much more aggressive in adding blind checks, to try to capture some points from all of those flaws. Our conservative rationale had been that retaining performance was more important, but with that many flaws per CB, the balance is changed. Even so, defensive rewriting earned FuzzBomb more points than its active analysis and repair capability. This result supports our notion that CGC-relevant flaws boil down to a small number of patterns in binary, and can be addressed with a small number of repair/defense strategies.

Fortunately, the story is not over for FuzzBomb; we have other customers who are interested in the technology, and we are actively pursuing transition opportunities to more real-world cyber defense applications.

Acknowledgments

This work was supported by DARPA and Air Force Research Laboratory under contract FA8750-14-C-0093. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

References

[1] D. J. Musliner et al., "Fuzzbuster: Towards adaptive immunity from cyber threats," in Proc. SASO-11 Awareness Workshop, October 2011, pp. 137–140.

[2] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Toronto, ON, Canada, Jul. 2011, pp. 12–22.

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

[4] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), ser. LNCS, vol. 4963. Springer, Apr. 2008, pp. 337–340.

[5] S. Edelkamp et al., "Survey on directed model checking," in Model Checking and Artificial Intelligence, 2008, pp. 65–89.

[6] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in Static Analysis Symposium (SAS), Venice, Italy, Sep. 2011, pp. 95–111.

[7] M. Helmert, "The fast downward planning system," Journal of Artificial Intelligence Research, vol. 26, no. 1, 2006, pp. 191–246.

[8] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 1083–1094. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568293

[9] J. Antunes, N. Neves, and P. Verssimo, "Reverse engineering of protocols from network traces," in Proc. 18th Working Conf. on Reverse Engineering (WCRE), 2011, pp. 169–178.

[10] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog," in NDSS, 2006.

[11] S. E. Friedman and D. J. Musliner, "Automatically repairing stripped executables with CFG microsurgery," in Adaptive Host and Network Security Workshop at the IEEE Int'l Conf. on Self-Adaptive and Self-Organizing Systems, 2015.