# FUZZBOMB : Fully-Autonomous Detection and Repair of Cyber Vulnerabilities

David J. Musliner, Scott E. Friedman, Michael Boldt, J. Benton, Max Schuchard, Peter Keller
Smart Information Flow Technologies (SIFT) Minneapolis, USA
{dmusliner,sfriedman,mboldt,jbenton,mschuchard,pkeller}@sift.net

Stephen McCamant
University of Minnesota, Minneapolis, USA
mccamant@cs.umn.edu

*Abstract*—**SIFT and the University of Minnesota teamed up to create a fully autonomous Cyber Reasoning System to compete in the DARPA Cyber Grand Challenge. 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, including a content-agnostic binary rewriting system called BINSURGEON. We then review FUZZBOMB's performance in the first phase of the Cyber Grand Challenge competition.**

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

## I. INTRODUCTION

In June 2014, DARPA funded seven teams to build autonomous Cyber Reasoning Systems (CRSs) to compete in the DARPA Cyber Grand Challenge (CGC). SIFT and the University of Minnesota together formed the FUZZBOMB team [1], building on our prior work on the FUZZBUSTER cyber defense system [2], [3], [4] and the FuzzBALL symbolic analysis tool [5], [6], [7].

SIFT's FUZZBUSTER system was built to automatically find flaws in software using symbolic analysis tools and fuzz testing, refine its understanding of the flaws using additional testing, and then synthesize *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 [8]. 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 the University of Minnesota 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 [5].

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, we outline several of the major advances we developed for FUZZBOMB, including a new content-agnostic binary rewriting system called BINSURGEON. We discuss the technical advances that allow BINSURGEON's template-based rewriting of stripped binaries to mitigate vulnerabilities. Finally, we review FUZZBOMB's performance in the qualifying round of the CGC competition, and discuss lessons learned.

## II. BACKGROUND

### A. 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"). Each CB contains 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.

### B. FUZZBUSTER

Since 2010, we have been developing FUZZBUSTER [9] under DARPA's 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.

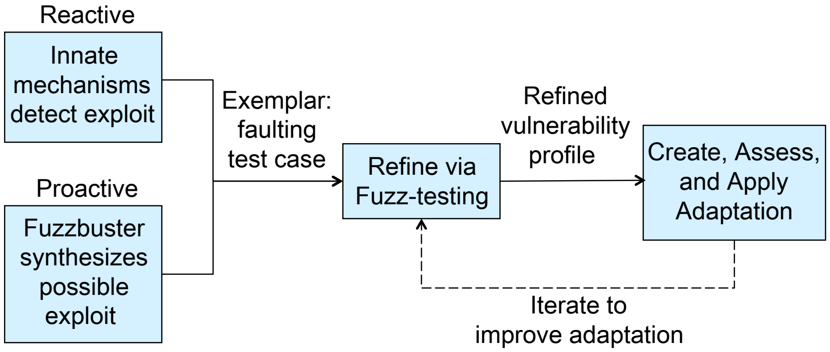

Reactive
Innate mechanisms detect exploit
Proactive
Fuzzbuster synthesizes possible exploit
Exemplar: faulting test case
Refine via Fuzz-testing
Refined vulnerability profile
Create, Assess, and Apply Adaptation
Iterate to improve adaptation

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

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. Proactive exemplars based on program analysis may be more informative: they can represent not just a single faulting input, but a set of constraints that define vulnerability-triggering inputs. Reactive exemplars pose a greater threat, since they almost certainly indicate that an attacker has already found a software flaw.

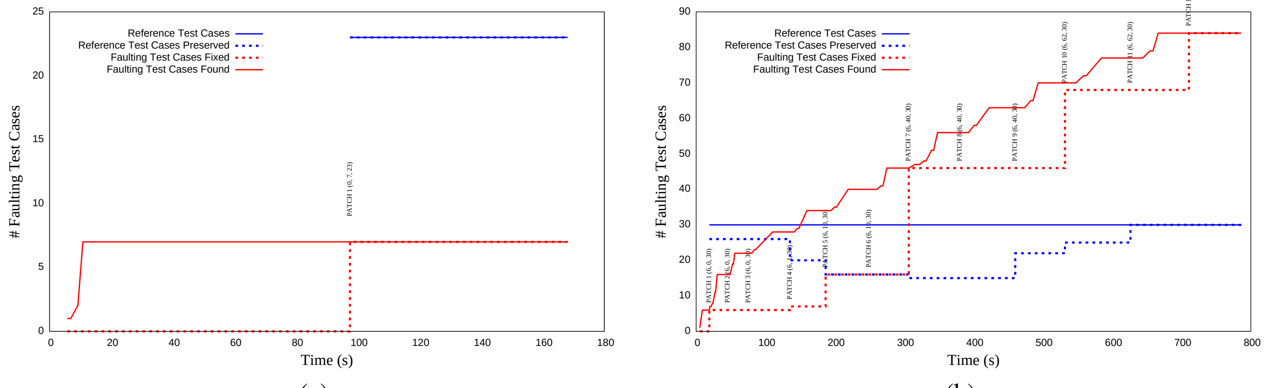
Starting from an exemplar, FUZZBUSTER uses its program analysis tools and fuzz-testing tools to refine its understanding of the vulnerability, building a *vulnerability profile* (VP). For example, FUZZBUSTER can use concolic testing to find that the long-string reactive exemplar is triggering a buffer 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. Similarly, constraint relaxation can generalize symbolic analysis exemplars to find additional paths to a vulnerability.

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. As it creates and applies adaptations, FUZZBUSTER can choose to re-evaluate previous adaptations, keeping those that remain effective and replacing those that have been superceded. FUZZBUSTER already has sophisticated techniques for creating filters that eliminate vulnerability-triggering inputs, which can be used as network-layer filters or application wrappers.

As different adaptations are developed, FUZZBUSTER can assess their performance against the set of tests it has been accumulating for a particular application, determining how effectively each adaptation stops known faulting inputs and preserves the functionality of known non-faulting test cases (either observed in the wild or generated by FUZZBUSTER) [10][1]. For example, Figure 2 illustrates FUZZBUSTER's performance on two applications, showing how it finds vulnerabilities (indicated by faulting test cases, the solid red line) and creates adaptations (patches) that try to fix those faults. The dotted red line indicates the number of faulting test cases that no longer cause a fault in the patched application. We refer to the undesirable area between those red lines, during which known vulnerabilities are still exploitable, as the *exposure*.

The blue lines show the performance of the original application (solid blue) and patched application (dotted) on the non-faulting test cases. In the first example, Figure 2a, FUZZBUSTER's analysis of the detected flaw is perfect: its first patch fixes all the known faulting test cases and does not degrade performance on the reference test cases. In the second example, Figure 2b, FUZZBUSTER creates a series of different patches and filters to shield a large number of different faulting inputs, and in the process, some of those degrade the application's performance on the non-faulting test cases (i.e., a gap appears between the solid and dotted blue lines). However, eventually FUZZBUSTER replaces the lesser adaptations with highly refined adaptations that restore all of

[1] We call this "poor man's regression testing," since it does not require any manually-created regression tests.



(a)

(b)

Figure 2. FUZZBUSTER works continuously to derive better adaptations, improving an application's performance on faulting and non-faulting test cases.

the performance and still prevent exploitation of all the known vulnerabilities.

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 in Section III.

### C. 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 SMT solver [11], [12] (such as STP [13] or Z3 [14]) 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.

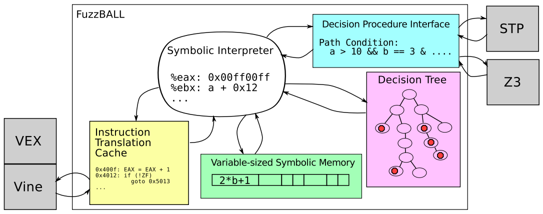


Figure 3. An overview of our FuzzBALL binary symbolic execution engine.

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 3 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.

To factor out instruction-set complexity, FuzzBALL builds on the BitBlaze Vine library [15] for binary code analysis, which provides a convenient intermediate language (the "Vine IL") for representing instruction behavior. Another complexity that arises at the binary level is that because memory is untyped, loads may not have the same size and alignment as stores. For example, a location might be written with a 4-byte store and then read back with a sequence of 1-byte loads. FuzzBALL optimizes for the common case by representing symbolic values in memory at the granularity with which they were stored, if they are naturally aligned, using a tree structure. But it will automatically insert bitwise operations to subdivide or assemble values as needed.

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. Babić *et al.* [5] combined dynamic control-flow analysis, static memory access, and FuzzBALL to find test cases for buffer overflow vulnerabilities in binaries, using the results of static analysis to guide FuzzBALL's search toward potential vulnerabilities. Martignoni *et al.* [7] used FuzzBALL to generate high-coverage test cases for CPU emulators, illustrating how exhaustive exploration is feasible for small but critical code sequences. Caselden *et al.* [6] combined dynamic data-structure analysis and FuzzBALL to produce proof-of-concept exploits for vulnerabilities that are reached only after complex transformations of a program's input, using novel pruning and choice heuristics to efficiently find inverse images of transformations such as data compression.

For FUZZBOMB and the CGC, we integrated FuzzBALL with the FUZZBUSTER reasoning framework and significantly extended FuzzBALL's program analysis capabilities.

## III. 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. Fault detection methods include monitoring communication channels (sockets) for failure and watchdog processes that send periodic messages to ensure liveness. The Optimi are given unique integer identifiers, and the next-in-order Optimus becomes Prime if the prior OP is determined to have failed; handshake messages ensure that the other Optimi agree on the new OP selection. 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 government-supplied response location.

### 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. 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 (e.g.,

[16], [17]). For example, Edelkamp *et al.* [16] 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.* [18]).

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 [19] to direct symbolic execution, in a manner similar to call-chain backwards symbolic execution [18]. 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.

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, etc.— FuzzBALL now uses similar methods to detect 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 [20]. We completed our own first version of this capability, along with associated test cases and SMT heuristic improvements. However, as noted in Section VI, 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 [21]. 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 [22], [23], [24], 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.

## IV. BINARY REWRITING

Here we describe background on binary rewriting and related work to clarify the technical contribution of BINSURGEON, FUZZBOMB's binary rewriting subsystem.

### A. *Control flow graphs*

BINSURGEON operates on a binary's Control Flow Graph (CFG) to modify the binary. For the purposes of BINSURGEON, a CFG is comprised of assembly instructions grouped into *blocks* with exactly one entry point and one exit point. At the exit point of any block, the program either (a) transitions to the entry point of the adjacent block in memory, (b) transitions the entry point of another block via a *control flow instruction* such as jumps or calls, or (c) terminates. These blocks and the control flows between them comprises the nodes and edges, respectively, of a directed— and often cyclic— graph.

The executable's functions are subgraphs of the CFG, often bounded by called blocks at the source(s) and return blocks at the sink(s), but exceptions exist, e.g., due to uncalled (or indirectly called) functions and functions that conclude with program termination rather than return instructions. To account for these exceptions, BINSURGEON infers function subgraphs by searching forward from called blocks and searching backward from return blocks, merging the intersecting block-sets, and also using common compiler idioms to identify function prologues and epilogues.

CFGs are recovered by disassembling the binary, which is a potentially-unsound process, since it is undecidable whether bytes in a stripped binary correspond to data or code [25], [26]. This means that a smaller rewrite to the CFG is better, all else being equal, since it relies on less of the potentially-incorrect subgraph of the CFG.

Figure 4 shows a small CFG snippet of a single function "Original Fn" rewritten by BINSURGEON to produce "Padded Fn" and then "Cookied Fn," as we describe in more detail

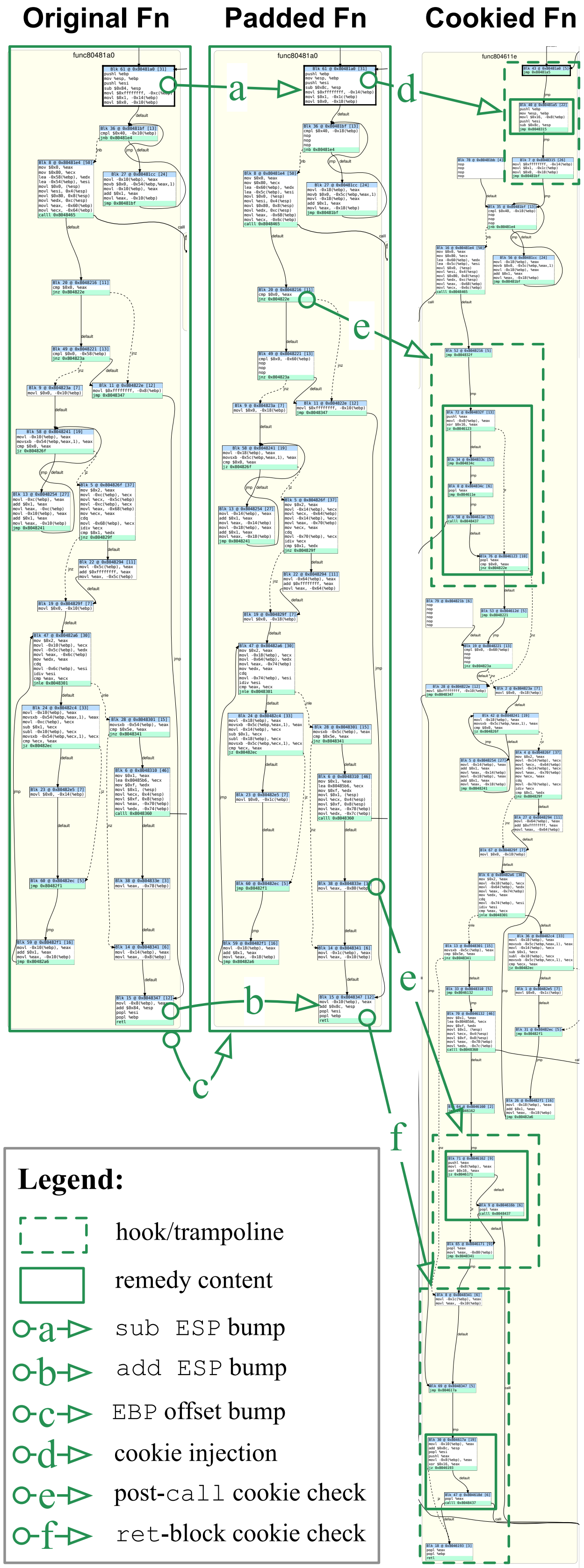


Figure 4. BINSURGEON rewrites a function to (1) add stack padding with space-preserving rewrites and (2) add a stack cookie with non-space-preserving rewrites.

in the next section. Instruction blocks are rendered as lists of instructions, with edges to subsequent blocks in the CFG. The shaded graphs are CFG subgraphs showing different revisions of the same function (original, padded, and cookied). Each directed edge from a version to the next indicates a single rewrite over the outlined blocks. The letter on each edge indicates the type of rewrite. For instance, there is one cookie injection (labeled "d") and two post-call cookie checks (labeled "e") written into the cookied function.

### B. Revising CFGs

We distinguish between two types of revisions to a CFG, both of which are supported by BINSURGEON:

1) *Space-conserving* rewrites replace or remove instructions from the CFG without requiring additional space, e.g., by reordering instructions or substituting an instruction for an instruction of equal byte-size.
2) *Space-consuming* rewrites modify the CFG in a way that requires additional space, e.g., by adding instructions to existing functions/blocks or addition new functions altogether.

These rewrites have an important practical difference: space-conserving rewrites will preserve the integrity of the unchanged CFG; but space-consuming rewrites require instructions to be shifted or relocated entirely, which potentially changes the size and byte representation of instructions (including relative control flow instructions). Space-consuming rewrites may thereby cause arbitrarily-large ripples in the CFG, so they require special attention.

One technique for implementing space-consuming rewrites is to write a *trampoline*, where a `jmp` instruction is written over the existing instructions, and the overwritten instructions— and others to be injected— are written to a blank space in the binary, which is targeted by the first `jmp` and terminates in a `jmp` back to the existing control flow.

In Figure 4, solid outlines around rewritten blocks indicate a space-conserving rewrite, and dashed outlines around rewritten blocks indicate space-consuming rewrites that required a trampoline.

### C. Related work in binary rewriting

Previous work has explored specialized binary rewriting to harden or diversify binaries. For instance, some rewriters perform targeted rewriting to inject single, specialized defenses such as stack cookies in return blocks [27] or control flow checks in return blocks or before indirect calls [28].

Many recent systems perform binary rewriting to increase diversity. *In-place code randomization* (IPCR) performs space-conserving rewrites to substitute and reorder instructions to help prevent code reuse attacks [29]. Similarly, *chronomorphic* programs perform space-conserving rewrites— including IPCR and block relocation— during their execution [30] to diversify themselves against code reuse attacks and cyber-reconnaissance (e.g., [31]). Other systems perform load-time binary rewriting to diversify binaries with a modified loader [26], [32]. These specialized rewriters locate blocks at randomized locations in memory and then ensure the CFG is intact.

TABLE I
OUTLINE OF BINSURGEON'S BINARY REWRITING PROCEDURE.

**GIVEN**: Set of insertions/deletions to the CFG.
*Compute the scope of the rewrite:*

- **SET** affected blocks $B$ = blocks that will change content.
- **SET** frontier blocks $F = B$.
- **WHILE** any block $f \in F$ is too small to hold a `jmp` instruction, add $f$'s source block(s) to $F$ and $B$; remove $f$ from $F$.

*Label the graph and rewrite it:*

- **CLAIM** all space presently occupied by $B$ as freespace.
- **LABEL** every block in $B$ and every internal control flow instruction accordingly.
- **HOOK** control flow at the previous start addresses of all $F$ by writing labeled `jmp` instructions to their new labels.
- **REWRITE** the labled graph in memory with the insertions and deletions.

*Inject the rewritten, labeled subgraph back into the binary:*

- **ASSEMBLE** instructions to estimate their size in the binary.
- **PACK** instructions into freespaces.
- **TEST** the packing job by assembling a custom linker script.
  - **IF** we overflowed a freespace:
    - ∗ **IF** other freespaces are above `jmp` size, update instruction size(s) accordingly and **GOTO: PACK**.
    - ∗ **ELSE** return *not-enough-space*.

*Repair* BINSURGEON*'s CFG model in memory:*

- **REMOVE** nodes corresponding to former blocks $B$ and all edges from those nodes.
- **ADD** nodes and incident edges for newly-assembled blocks $B^{SIFT}$.
- **SPLIT** blocks as necessary if new outward edges from $B^{SIFT}$ fall between a block's entry and exit points.

Other methods exist for translating binaries into an intermediate representation (IR) (e.g., [33], [34]), and then rewriting them back into machine code, e.g., for diversity or safety purposes. In contrast to IR approaches, BINSURGEON rewrites the CFG and assembly instructions directly, which avoids potential IR translation errors and potential performance degradation by making local, targeted changes. As we demonstrate in the next section, the CFG and assembly instructions themselves are expressive enough to write diverse templates for program repair and defense.

Other tools such as DynInst[2] automatically instrument the binary, but they consume substantially higher disk space, memory footprint, or performance overhead. For example, DynInst's instrumentation has been shown to increase runtime overhead by 96% [35]; that performance penalty would have led to zero scores in the CGC.

BINSURGEON's rewrites are far less invasive and costly: BINSURGEON adds no universal function call hooks or virtualization, so the overhead of its modifications is only proportional to the specific installed defenses/repairs.

[2] http://www.dyninst.org/

## V. Repair and Defense with BinSurgeon

Here we overview BinSurgeon's procedure for rewriting stripped, third-party binaries to add or remove arbitrary content [36]. We then describe some binary rewriting templates that BinSurgeon uses for program defense and repair as part of FuzzBomb.

FuzzBomb's binary rewriting algorithm is summarized in Table I. The procedure is given a CFG and a set of insertions and/or deletions to the CFG. The insertions and deletions are specified relative to existing instructions in the CFG (e.g., *insert instructions X before instruction y* or *delete instructions Z*). BinSurgeon does *not* use absolute addresses (e.g., *insert instructions X at address y*) for insertions and deletions, since making space-consuming changes could shift the addresses of subsequent instructions, thereby invalidating other absolute addresses.

BinSurgeon's rewriting procedure first identifies *affected* blocks that must be rewritten and relocated, as well as *frontier* blocks that will connect the affected blocks to the rest of the CFG. The affected blocks will be rewritten, and if BinSurgeon overflows these blocks, it will utilize (or append) remote *freespace* (i.e., available executable memory) within the binary. BinSurgeon identifies frontier blocks iteratively, since not all blocks are large enough to support `jmp` instructions (i.e., for a trampoline, described in Section IV-B). The frontier blocks serve as trampoline `jmp` sites for the affected blocks, which is the trampoline content.

After identifying affected and frontier blocks, BinSurgeon *labels* these blocks from their absolute addresses by injecting assembly labels before each block, and then it rewrites all internal control flow edges (i.e., conditional or unconditional jumps between affected blocks $b_1 \in B$ and $b_2 \in B$) to use these labels. BinSurgeon writes `jmp` instructions at the former entry point of each frontier block to build a compound trampoline into the labled affected blocks. BinSurgeon does not explicitly write `jmp` instructions *back* to the unmodified CFG; rather, it uses the existing control flow instructions of the labled blocks, which will be reassembled later in its procedure. It then rewrites the labeled, labled graph with the given insertions and deletions.

BinSurgeon next injects the rewritten, labled graph back into the binary, using the affected blocks' previous locations— and other claimed/extended executable memory— as freespace. This is a greedy, iterative process of instruction-packing: BinSurgeon finds the next freespace proximal to the last freespace (since near `jmp` instructions require fewer bytes) and writes as many instructions as possible, insofar as it can also write a `jmp` instruction to the next freespace.

After packing its freespaces, BinSurgeon writes out a custom linker script to assemble all of the desired instructions at the desired addresses. This converts every instruction of the labled CFG subgraph into the machine-executable, location-specific opcodes. If the assembling and linking succeeds, BinSurgeon writes the corresponding instruction bytes directly into the binary and reports success.

TABLE II
Remedies implemented by BinSurgeon for FuzzBomb

*Support* remedies add utilites for defense & repair:

- `cleanup`: substitutes instructions in the CFG with instructions guaranteed to re-assemble.
- `add-text-section`: appends a new executable section to the binary by extending or adding a program header.
- `fn-inject`: adds new function(s) to the binary.
- `fn-intercept`: intercepts existing functions by rerouting direct calls to new or existing functions.
- `add-data-space`: adds space in the binary for static data storage.

*Repair* remedies address known PoVs:

- `terminate`: injects instruction(s) to terminate the program at the PoV location.
- `o/w-terminate`: overwrite existing instructions to terminate the program at the PoV location.
- `null-ptr-check`: test a register or memory address, and terminate if zero.
- `stack-top-cookie`: write a cookie value to the top of the program stack. Check it at the PoV location; terminate if overwritten.
- `heap-cookie`: intercept malloc, write a cookie value after each allocation. Check it at the PoV location; terminate if overwritten.
- `bss-cookie`: write cookie value(s) into the binary's static data segment. Check it at the PoV location; terminate if overwritten.

*Repair & Defense* addresses known/unknown vulns:

- `stack-pad`: increase stack frame size; decrement all base pointer offsets below a given threshold.
- `stack-cookie`: write a constant to frame pointer between local variables or before the return address. Check the cookie upon return or after function calls; terminate if overwritten.
- `range-check`: if a memory address (e.g., pointer or function pointer) is not within a given range (e.g., text section), terminate.
- `receive-check`: intercept input functions and terminate if they will write to illegal memory ranges.
- `cfi`: range-based control flow integrity on return addresses and indirect `call` and `jmp` addresses.

In some cases, the assembled instructions may overflow a freespace. This occurs when BinSurgeon underestimates instruction sizes and thereby over-packs a freespace. In these cases, BinSurgeon updates its size estimates and attempts to re-pack in the remaining freespaces. Otherwise, if it has no more freespace, BinSurgeon reports that it needs more space.

Finally, BinSurgeon repairs its in-memory model of the program CFG, since the insertions and deletions may well have changed existing functions and blocks connectivity or added new functions and blocks altogether.

BinSurgeon's rewriting procedure is *content agnostic*, which means its rewriting *capability* is decoupled from the rewritten *content*. As a practical consideration, this allowed us to develop BinSurgeon independently of the repair and defense templates it deployed for FuzzBomb.

### *A. Repairing & Defending Binaries*

BinSurgeon uses rewriting templates— which we call *remedies*— to harden and repair binaries. Figure 5 shows a

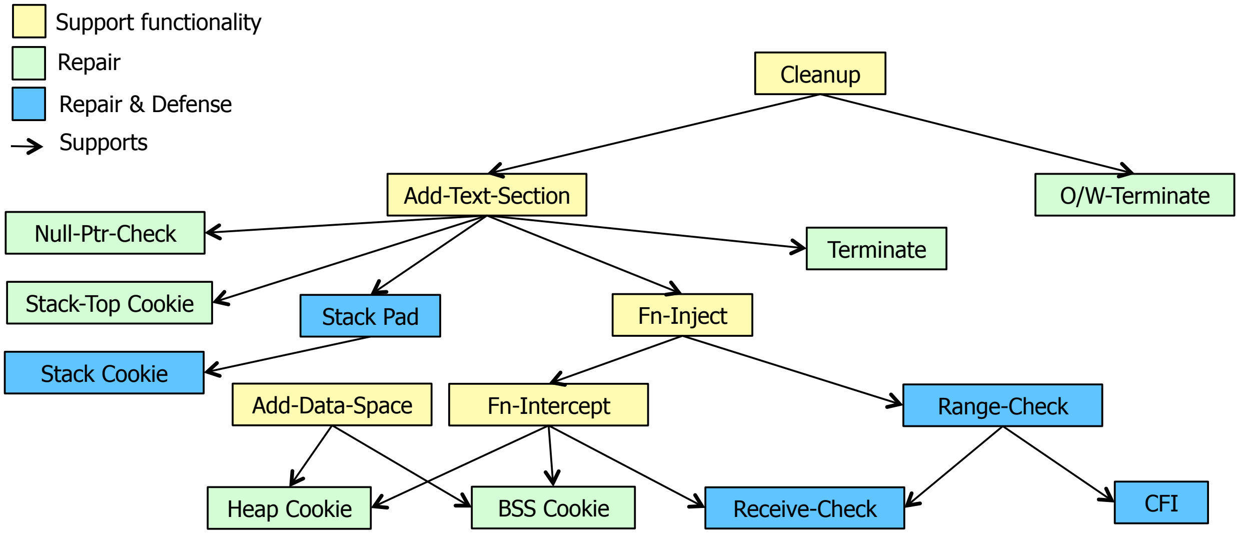


Figure 5. Remedies for templated binary rewriting, including support functionality, targeted repair templates, and defensive templates.

dependence graph of remedies, since some remedies depend on others' functionality, and Table II lists a brief description of each remedy. Each remedy takes one or more parameters (e.g., a vulnerable function or instruction) and produces a set of instruction insertions and deletions to use with BINSURGEON's rewriting procedure.

These specific remedies are designed to avoid compromised states or terminate the program when a compromised state exists. Intuitively, when the program is in a compromised state— or in program states where compromise is imminent and unavoidable— terminating the program safely is preferable to relinquishing control to a cyberattack.

These remedies do not fix the *underlying* problems, such as overflows or off-by-one errors; rather, they mitigate the adverse, exploitable manifestations. Templated repair of the underlying problems are the focus of some source-code repair systems (e.g., [37]), which is evidence that we can also develop BINSURGEON templates to fix underlying problems if they are adequately described. Next, we describe some novel and/or counter-intuitive remedies in additional depth.

The simplest remedies are `terminate` and `o/w-terminate`, which terminate the program at a specified location in the CFG. The `o/w-terminate` (overwrite) remedy does this without first allocating freespace, in case the binary cannot be properly extended.

The `stack-pad` and `stack-cookie` remedies are used in succession to protect a function's stack frame by (1) adding padding to a stack frame before or between the local variables, and (2) writing a *cookie* value within that padding, to flag an overflow if it is overwritten. Figure 4 illustrates the injections and deletions specified by these remedies as performed by BINSURGEON: `stack-pad` (Figure 4, middle) revises the setup and reset of the stack frame (Figure 4 [a] and [b], respectively) and revises all references to the stack via the base pointer (Figure 4[c]); and `stack-cookie` (Figure 4, right) injects a cookie at the head of the function (Figure 4[d]), and adds cookie checks after each function call (Figure 4[e]) and at the return block (Figure 4[f]).

One of the most complex remedies used within FUZZBOMB is the `heap-cookie`. This remedy template is comprised of the following modifications:

1) Injecting functions that intercept memory management functions, e.g., `malloc` and `free`, that allocate and free an extra byte, respectively, and write a specific value to the extra byte, and store the location of the byte within an injected array.
2) Overwriting `call` instructions to `malloc` and `free` to instead invoke the injected functions.
3) Inject a cookie-checking function that iteratively checks the cookie array, and terminates if any have changed value.
4) Inject a call to the cookie-checking function at the location of the PoV.

In conjunction, these modifications to the CFG cause the program to add an extra cookie-byte to each heap allocation and then check these cookie-bytes where specified, terminating if it senses an overwrite.

## 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 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 re-allocate 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 extracted protocol had no variables to fuzz. And the default fuzz-testing patterns were not used because there *was* a protocol extracted. Thus FUZZBOMB's fuzzing was completely disabled for all of the challenge problems. Also, because the re-allocation functionality was not available, we had to pre-allocate 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 expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

## REFERENCES

[1] D. J. Musliner, S. E. Friedman, M. Boldt, J. Benton, M. Schuchard et al., "Fuzzbomb: Autonomous cyber vulnerability detection and repair," in Proceedings INNOV 2015: The Fourth International Conference on Communications, Computation, Networks and Technologies, 2015.

[2] D. J. Musliner, J. M. Rye, D. Thomsen, D. D. McDonald, M. H. Burstein et al., "Fuzzbuster: Towards adaptive immunity from cyber threats," in Proc. SASO-11 Awareness Workshop, October 2011.

[3] ——, "Fuzzbuster: A system for self-adaptive immunity from cyber threats," in Proc. Eighth Int'l Conf. on Autonomic and Autonomous Systems, March 2012.

[4] 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 Int'l Conf. on Self-Adaptive and Self-Organizing Systems, September 2012.

[5] 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, July 2011.

[6] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "HI-CFG: Construction by binary analysis, and application to attack polymorphism," in ESORICS'13: European Symposium on Research in Computer Security, London, UK, Sep. 2013.

[7] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), London, UK, Mar. 2012.

[8] 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.

[9] S. E. Friedman, D. J. Musliner, and J. M. Rye, "Improving automated cybersecurity by generalizing faults and quantifying patch performance," International Journal on Advances in Security, vol. 7, no. 3-4, 2014, pp. 121–130.

[10] D. J. Musliner, S. E. Friedman, T. Marble, J. M. Rye, M. W. Boldt et al., "Self-adaptation metrics for active cybersecurity," in Proc. Adaptive Host and Network Security Workshop at the IEEE Int'l Conf. on Self-Adaptive and Self-Organizing Systems, September 2013.

[11] S. Ranise and C. Tinelli, "The SMT-LIB format: An initial proposal," in Pragmatics of Decision Procedures in Automated Reasoning (PDPAR), Miami, FL, USA, Jun. 2003.

[12] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL($t$)," J. ACM, vol. 53, no. 6, 2006, pp. 937–977.

[13] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in Computer Aided Verification (CAV), Berlin, Germany, Jul. 2007.

[14] 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.

[15] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager et al., "BitBlaze: A new approach to computer security via binary analysis," in Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper., Hyderabad, India, Dec. 2008.

[16] S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs, A. Fehnker et al., "Survey on directed model checking," in Model Checking and Artificial Intelligence, 2008, pp. 65–89.

[17] J. Benton, A. J. Coles, and A. Coles, "Temporal planning with preferences and time-dependent continuous costs," in International Conference on Automated Planning and Scheduling, 2012.

[18] 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.

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

[20] 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

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

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

[23] W. Cui, J. Kannan, and H. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. USENIX Association, 2007, pp. 1–14.

[24] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in 15th Symposium on Network and Distributed System Security (NDSS), 2008.

[25] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in Machine Learning and Knowledge Discovery in Databases. Springer, 2011, pp. 522–536.

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

[27] A. Baratloo, N. Singh, T. K. Tsai et al., "Transparent run-time defense against stack-smashing attacks." in USENIX Annual Technical Conference, General Track, 2000, pp. 251–262.

[28] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in USENIX Security, 2013, pp. 337–352.

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

[30] S. E. Friedman, D. J. Musliner, and P. K. Keller, "Chronomorphic programs: Runtime diversity prevents exploits and reconnaissance," International Journal on Advances in Security, vol. 8, no. 3-4, 2015, pp. 120–129.

[31] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in Proceedings of the 35th IEEE Symposium on Security and Privacy, 2014.

[32] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in Network and System Security. Springer, 2013, pp. 293–306.

[33] P. Anderson and M. Zarins, "The codesurfer software understanding platform," in Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on. IEEE, 2005, pp. 147–148.

[34] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: a binary analysis platform," in Computer aided verification. Springer, 2011, pp. 463–469.

[35] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in Proc. IEEE Int'l Symp. on Performance Analysis of Systems and Software, 2010.

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

[37] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 802–811.