Automatically Repairing Stripped Executables with CFG Microsurgery

Scott E. Friedman and David J. Musliner
Smart Information Flow Technologies (SIFT)
Minneapolis, MN, USA
Email: {sfriedman, dmusliner}@sift.net

***Abstract*—BINSURGEON is a binary rewriting system that enhances stripped binary executables with repairs, defenses, and additional functionality. This involves making space-consuming changes to the program's control flow graph (CFG), recomputing instruction content, and relocating instructions, all while preserving functionality in the remainder of the program's control flow. BINSURGEON uses extendable rewrite templates that enable other systems to specify and parameterize program modifications, which allows BINSURGEON to be a fully-automatic component of a larger system. In this paper, we describe BINSURGEON in the context of the FUZZBOMB automated program analysis and repair system. We outline BINSURGEON's general binary rewriting algorithm for modifying CFGs according to FUZZBOMB's rewrite templates. We also review some of FUZZBOMB's rewrite templates to demonstrate the diverse repair and defense strategies— including stack protection, heap protection, CFI, pointer-checking, and more— that are implemented by BINSURGEON to harden and repair vulnerable binaries.**

Keywords-binary rewriting, cybersecurity, program repair.

I. INTRODUCTION

Cyberattacks threaten today's computer systems, with intrusions increasing in complexity and frequency every year [1], [2]. We can counter this threat by improving the programs at enterprise perimeters and thereby reducing the cyber attack surface. Unfortunately, this presents multiple practical challenges:

- Manually repairing vulnerable programs requires expertise and familarity with the source code.
- Administrators must verify, prioritize, and then sequentially address bug reports.
- Source code is not always available for third-party or legacy executables.

Even with an army of expert programmers (and source code) available for a trivial vulnerability, the perfect fix may take tens of minutes to complete, test, and deploy, which is orders of magnitude longer than the duration of a typical transaction-based ROP attack.

We developed FUZZBOMB under DARPA's Cyber Grand Challenge (CGC) program to automatically analyze and repair vulnerabilities in programs. FUZZBOMB uses the FuzzBALL symbolic execution engine [3], [4], [5] to detect vulnerabilities within the binary, and then it performs binary rewriting to revise the executable on disk to defend it against exploitation. This paper describes FUZZBOMB's automatic binary rewriting system, called BINSURGEON. BINSURGEON takes a stripped binary and an optional vulnerability report as input, and it automatically rewrites the binary with multiple strategies to produce repaired or hardened variants. BINSURGEON does all of this in the order of seconds.

Hardening a program may come at a cost: for instance, if the *control flow graph* (CFG)— which statically describes the program's instructions and execution paths— is incomplete or incorrect, then binary rewriting may produce erroneous results. For this purpose, FUZZBOMB optionally runs a regression test suite to score and rank the BINSURGEON variants. If a regression test suite is not provided, FUZZBOMB uses the FuzzBALL symbolic execution engine to accumulate test cases.

In this paper, we distinguish between two types of binary rewrites:

1) *Repair*: Rewrite the binary to repair a known *proof of vulnerability* (PoV) faulting test case. This is reactive behavior.
2) *Defense*: Rewrite the binary to improve safety, in the absence of a PoV. This is proactive behavior.

BINSURGEON performs both types of binary rewrites using the same underlying technology; the only difference is the *content* of the rewrites and the presence or absence of a PoV. It does all of this by directly modifying the assembly instructions in the program's CFG. FUZZBOMB recovers the CFG from the binary automatically, using mixed recursive/-linear disassembly of the binary, static identification of jump tables, and dynamic tracing to identify indirect control flow (e.g., jump addresses stored as data) [3]. This means that BINSURGEON does not use any intermediate languages or reverse compilation techniques (cf. [6], [7]).

This paper describes BINSURGEON, which we have used to rewrite 32-bit x86 CGC binaries and ordinary x86 Linux executables [8]. FUZZBOMB is a working demonstration that BINSURGEON's binary rewriting technology is an effective component in an autonomous vulnerability repair system. Furthermore, BINSURGEON is ready for integration with future autonomous systems that adapt binaries with *non*-defensive content (e.g., functionality patches) or for human-guided binary rewriting (e.g., for mixed-initiative cyberdefense).

We begin by outlining the representations and inputs to BINSURGEON, including background material and related

work. We then review BINSURGEON's rewriting algorithm and provide examples of BINSURGEON's rewriting templates for repair and defense, as used within FUZZBOMB. We close with a discussion of opportunities to extend FUZZBOMB's rewriting templates and core rewriting capability.

II. BACKGROUND

Here we describe background on binary rewriting and related work to contextualize BINSURGEON's contribution.

A. Control flow graphs

As mentioned above, BINSURGEON operates on a binary's CFG in order 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.

Figure 1 shows a small CFG snippet of a single function (at left) rewritten twice (middle and right) by BINSURGEON, as we discuss later.

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 [9], [10]. This means that a smaller rewrite to the CFG is better, all else being equal, since it assumes less of the potentially-incorrect subgraph of the CFG.

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.

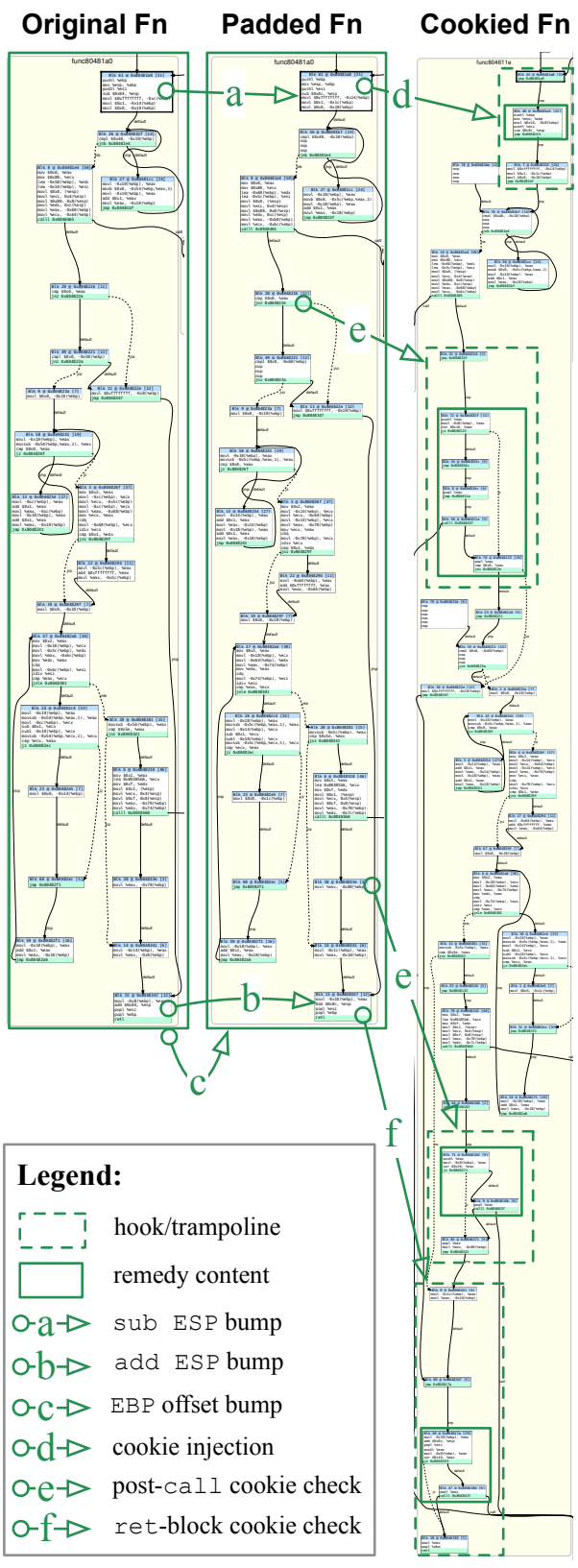

Figure 1. BINSURGEON rewrites a function to (1) add stack padding with space-preserving rewrites and (2) add a stack cookie with non-space-preserving rewrites.

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.

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 [11] or control flow checks in return blocks or before indirect calls [12].

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 [13]. Similarly, *chronomorphic* programs perform space-conserving rewrites— including IPCR and block relocation— during their execution [8] to diversify themselves against code reuse attacks and cyber-reconnaissance (e.g., [14]). Other systems perform load-time binary rewriting to diversify binaries with a modified loader [10], [15]. These specialized rewriters locate blocks at randomized locations in memory and then ensure the CFG is intact.

Other methods exist for translating binaries into an intermediate representation (IR) (e.g., [6], [7]), 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[1] automatically instrument the binary, but they consume substantially higher disk space, memory footprint, or performance overhead. BINSURGEON's rewrites are less invasive by comparison, since its present operating setting (DARPA CGC) imposes tight limits on disk, memory, and performance.

III. REWRITING WITH BINSURGEON

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

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

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'.
- **SPLIT** blocks as necessary if new outward edges from B' fall between a block's entry and exit points.

A. A Content-Agnostic CFG Rewriting Procedure

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

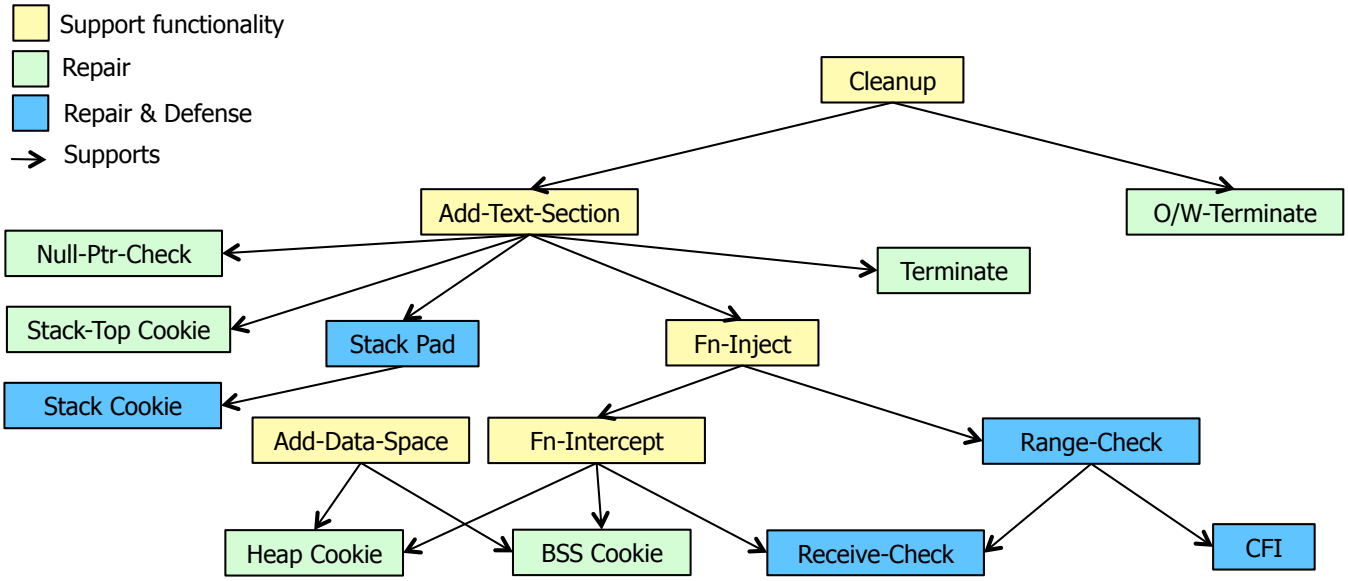

Figure 2. Remedies for templated binary rewriting, including support functionality, targeted repair templates, and defensive templates.

Section II-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.

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. We discuss these repair and defense rewrites next, to demonstrate the practicality of BINSURGEON.

B. Repairing & Defending Binaries

Here we describe rewriting templates— which we call *remedies*— that BINSURGEON uses within FUZZBOMB to harden and repair binaries. Figure 2 shows a 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 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., [16]), 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 1 illustrates the injections and deletions specified by these remedies as performed by BINSURGEON: `stack-pad` (Figure 1, middle) revises the setup and reset of the stack frame (Figure 1 [a] and [b], respectively) and revises all references to the stack via the base pointer (Figure 1[c]); and `stack-cookie` (Figure 1, right) injects a cookie at the head of the function (Figure 1[d]), and adds cookie checks after each function call (Figure 1[e]) and at the return block (Figure 1[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.

IV. CONCLUSION AND FUTURE WORK

We have presented the BINSURGEON general binary rewriting system. We distinguished BINSURGEON's binary rewriting *capability* (i.e., its general rewriting procedure) from the *content* that it writes into the binary. We described BINSURGEON's rewriting capabilities in the setting of the FUZZBOMB fully-automated binary analysis and repair system, including template-based rewrites that detect and react to corrupton of stack memory and heap memory.

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.

As mentioned above, FUZZBOMB presently uses BINSURGEON to mitigate the *effect* of vulnerabilities in order to prevent subsequent malicious exploits; however, we believe that the same template approach can be used to fix the vulnerability itself (e.g., integer overflow, off-by-one-error, buffer overflow), provided information about the problem. This is already being done with templates at the source code level (e.g., [16]), so constructing templates in a stripped binary setting is an area of future work on FUZZBOMB and BINSURGEON technologies.

We do not regard FUZZBOMB's set of template-based repairs and defenses as a complete set of strategies for hardening binaries, but it illustrates BINSURGEON's diverse capabilities. In the near term, we plan to increase FUZZBOMB's set of program-hardening templates, e.g., with

different CFI strategies (e.g., [12]) and memory corruption detectors.

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. Approved for Public Release, Distribution Unlimited.

REFERENCES

[1] T. Kellerman, "Cyber-threat proliferation: Today's truly pervasive global epidemic," *Security Privacy, IEEE*, vol. 8, no. 3, pp. 70 –73, May-June 2010.

[2] G. C. Wilshusen, "Cyber threats and vulnerabilities place federal systems at risk: Testimony before the subcommittee on government management, organization and procurement," United States Government Accountability Office, Tech. Rep., May 2009.

[3] 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.

[4] 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. [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/caselden13esorics.pdf

[5] 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.

[6] 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.

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

[8] S. E. Friedman, D. J. Musliner, and P. K. Keller, "Chronomorphic programs: Using runtime diversity to prevent code reuse attacks," in *Proceedings ICDS 2015: The 9th International Conference on Digital Society*, Feb. 2015.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.