A Smart Fuzzer for x86 Executables

Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, Roberto Paleari
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Milano, Italy
{andrew,lorenzo,monga,roberto}@security.dico.unimi.it

Abstract

The automatic identification of security-relevant flaws in binary executables is still a young but promising research area. In this paper, we describe a new approach for the identification of vulnerabilities in object code we called smart fuzzing*. While conventional fuzzing uses random input to discover crash conditions, smart fuzzing restricts the input space by using a preliminary static analysis of the program, then refined by monitoring each execution. In other words, the search is driven by a mix of static and dynamic analysis in order to lead the execution path to selected corner cases that are the most likely to expose vulnerabilities, thus improving the effectiveness of fuzzing as a means for finding security breaches in black-box programs.*

1 Introduction

Fuzz testing [16] (also known as *fuzzing*) is a very simple technique of black-box testing. The basic idea is to attach the inputs of a program to a source of random data: any resulting failure is taken as a symptom of a defect in the program. This approach was successfully used to discover bugs in popular programs, though released without the source code [16, 15]. Since its introduction, the fuzzing approach has became a major tool in security analysis. However, while this approach works well when aimed at searching generic failures, penetration testers are usually interested in more specialized corner cases[1]. In fact, while generic failures might be exploited to cause denials of service, in order to find more subtle bugs suitable for program sabotage more control about the coverage of program paths is needed. By using random data, though, even a single equality condition on just one byte of the input data could be difficult to satisfy wittingly: in fact, satisfying randomly a 32-bit equality corresponds to guessing by chance the correct value out of four billion possibilities. Blind guessing is very unlikely to be successful and some clue about the structure and behavior of the application is needed. In a previous work of our group, focused on searching for buffer overflow vulnerabilities, signal monitoring was proposed to drive the fuzzing and exploiting process [4]. Recently, great progress has been achieved in the analysis of binary code, thus it seems sensible to leverage in these results to ease the search for nasty inputs that could be used to subvert an application by exploiting memory errors.

Program analysis can be conducted according to two basic approaches: (I) *static* or (II) *dynamic*. Static analysis techniques analyze the executable without actually running it: the program code is analyzed by reasoning over *all* the possible behaviors that might arise at run time. Static analysis is therefore *complete* (i.e. results are correct, no matter on what inputs the program is run), but often overly *conservative* since it normally reports properties weaker than the ones that actually hold in a specific execution. The conservative nature of static analysis produces several false positives. Conversely, dynamic analyses, while unable to examine all the possibilities, can provide more accurate information about the program data-flow, because the appropriate variable values in any particular execution states are know at run-time, thus avoiding the imprecision intrinsic in "aliasing" and indirection analysis.

In this paper we take a hybrid approach, in order to blend together the strength of both types of analysis [9]. To overcome the fundamental limitation of the original blind fuzzing, we propose to use a *smart fuzzing*, that exploits analysis techniques to improve its effectiveness in penetration testing.

The goal of our work is to discover inputs able to compromise the security of an application: such data are sometimes known as *inputs of death* [5]. The method we adopted can be divided into two distinct phases: (I) a preliminary static analysis needed for collecting information about the structure of the program

[1] A corner case is a problem or situation that occurs only outside of normal operating parameters – specifically one that manifests itself when multiple environmental variables or conditions are simultaneously at extreme levels [20].

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

and to identify interesting paths of execution; (II) a subsequent monitored execution of the program, in order to refine the knowledge about the program (e.g., by resolving the target of indirect control transfers, resolving pointers, ...) and to identify the corruption of sensitive program data (i.e., memory accesses and targets of indirect control transfers).

The knowledge collected during step (II) is refined at every monitored execution and used to drive the generation of the input for the next one. The idea is to choose the new input such that it forces the program to follow a given execution path. Analysis tries to reveal the dependencies among input data and the predicates controlling the execution of the observed paths, called *path conditions*, and their observed truth values. From these predicates we generate a high-level boolean formula (where the connections with the inputs are made explicit) and, using a constraint solver, we try to find what inputs we have to feed to the program to force it to follow the chosen path. Hopefully, after a bounded number of executions, if the program is vulnerable, the input chosen is able to trigger the vulnerability and we detect the corruption of one of the data we marked as sensitive.

The paper is organized as follows. In Section 2 we introduce a motivating example, and in Section 3 we present the framework we have developed. Section 4 describes the techniques adopted to analyze a program, to monitor its execution and to drive the input generation. In Section 5 we discuss some related works and, finally, in Section 6, we sketch the state of our prototype and draw some conclusions while discussing our plans for future work.

2 Motivating Example

The example[2] depicted in Figure 1 should make clear why blind fuzzing is often not very effective. The procedure `process_input` receives an array of characters and returns a sanitized copy, obtained by replacing every occurrence of the character `'\''` with the same character repeated twice. Since a temporary local buffer is used but its size is statically bounded, the procedure checks (line 5) whether the input can be securely copied into that buffer or not. In the latter case the procedure returns immediately. Despite this security check, the procedure is vulnerable to a stack-based buffer overflow. In fact an attacker might be able to invoke the procedure with a buffer filled with `'\''` and character duplication (lines 9–10) goes beyond the end of the local buffer possibly overwriting the return address stored into the stack.

It is easy to see that, even in a simple case like the one in the example, if inputs were chosen completely randomly the chances to force the program to execute all possible paths and to satisfy all the conditions necessary to trigger the vulnerability would be really small. Therefore, a smart fuzzing approach should leverage on the (partial) knowledge about the structure of the program. Our goal is to be able to preserve the ability to fuzz a program without knowing its source code (and hence its high level semantics). However, we suppose to be able to analyze the binary code in order to excerpt potential execution paths. Monitored executions should then be used to select class of inputs that are more likely to trigger interesting paths which could be vulnerable.

[2]In the following we deal with *machine code*, however, for the sake of clarity, the example is shown in its source code form.

```
1  char* process_input(char* buf) {
2    char newbuf[64];
3    int i, j = 0;
4
5    if (strlen(buf) <= 58) // good assumption!
6      for(i = 0; i < strlen(buf); i++) {
7        if (buf[i] >= 32 && buf[i] < 127) {
8           newbuf[j++] = buf[i];
9           if (buf[i] == '\'')
10             newbuf[j++] = buf[i];
11       }
12     }
13
14   newbuf[j] = '\0';
15
16   return strdup(newbuf);
17 }
```

Figure 1: C code of a vulnerable procedure.

3 Overview of the Framework

Our framework is aimed at crafting input data that are likely to drive a program into corner states suitable to memory attacks. Its use is iterative and self-refining: a program is executed several times using different inputs and the new information collected at every execution is used to add extra information in order to select the next input and to discover sensitive memory areas.

We start with traditional blind fuzzing, i.e., random input data and no *path conditions* ($\mathcal{PC} = \emptyset$). This initial program execution is monitored and whenever a conditional control transfer is encountered a data flow analysis is performed in order to determine if the predicate p used in the conditional statement depends on the input. If a dependency is found the path conditions are updated as follows: $\mathcal{PC}' = \mathcal{PC} \cup p$ is associated to the true path and $\mathcal{PC}'' = \mathcal{PC} \cup \neg p$ is associated to the false path. The process can be in principle repeated by monitoring the two forked execution paths and the path conditions are consequently updated. Path conditions can be used to select smart inputs for the subsequent fuzzing operations.

However, the above analysis is challenging and onerous when performed on source code [19] and its complexity is drastically increased when the program is available only in executable form: in fact, the assembled code has no high level predicates and the relation between assembly jump conditions (consisting only of elementary predicates) and higher-level expressions (needed for constraining input data) has to be reconstructed.

Our frameworks is organized in the following components:

- an *off-line analysis engine* which is used to perform an initial static analysis of the program to collect preliminary information, such as an approximation of the interprocedural control flow graph (ICFG) and loop identification, that will be refined in the next steps.
- An *on-line monitoring engine* which takes care of executing and monitoring the program and tracking dependencies between the input and the dynamic control transfers encountered.
- A set of *heuristic rules* is used to identify interesting paths that could lead to accesses to sensitive memory areas and potential security violations.
- A *constraint solver* that, given a path in the program we want to analyze and a set of path conditions we discovered through monitoring, returns an input that, when fed to the program, forces it to follow the chosen path.

4 Framework Architecture

4.1 Off-line analysis

The goal of this phase is to provide a conservative but potentially inaccurate view of the application that will then be refined during monitored execution. Thus, the executable object is parsed and disassembled using conventional techniques [6, 18]. In order to ease the analysis, machine instructions are expanded into a simple intermediate language that makes explicit side effects on registers and memory areas. A control flow graph of intermediate instructions is built. This step may lead to inaccurate results since control transfers in executable may be indirect, however we aim at producing an approximation whose accuracy will be improved by the on-line monitoring.

An important aspect of our approach is approximated loop analysis to reduce the need for indefinite iteration. A first part of the loop analysis process is performed off-line: the control flow graphs are processed in order to identify loops and the corresponding entry and exit points [2]. This step allows the on-line analyzer to detect when a new loop iteration is going to begin and when the execution is coming out from a loop construct.

```
1 mov    0xffffffb4(%ebp),%eax
2 add    0x8(%ebp),%eax
3 cmpb   $0x1f,(%eax)
4 jle    8
```

Figure 2: Assembly code generated by `gcc` for `buf[i] >= 32`.

4.2 On-line Analysis

Initially the program is executed with a random input data and the initial state of the program is collected. This *process startup state* ($\mathcal{PSS}$) consists of: the original state of processor registers, the set of allocated memory regions, the program arguments, the program environment and the list of dynamically loaded libraries. Further inputs received by the program during its execution (e.g. from a socket or a file) are added later, by analyzing the parameter of the system calls (e.g., `read`) executed by the program. The program execution is monitored using the debugging API provided by the operating system.

4.2.1 Data-flow Analysis

In order to track the information flow generated by input data we perform a dynamic data-flow analysis.

As the program is executed, every machine instruction processed is semantically expanded (using the intermediate form generated off-line) and for each new intermediate instruction i we compute the set $define(i)$, of defined variables, and $use(i)$, of used variables (the term variable is here used to describe both memory locations and registers). As we are working at run-time, we can precisely tell which are the memory locations used and defined by an instruction because we actualize all memory locations. For example the 8-bit memory location with address `EAX + EBX`, that in our intermediate representation is written as `m8[r32(EAX) + r32(EBX)]`, is treated as `m8[c32(80482f6)]`, where `80482f6` is the actual result of `r32(EAX) + r32(EBX)`. Working with actualized memory locations allows to precisely compute data-flow dependencies with no need of conservative overestimation.

The monitored program execution trace is constantly recorded in an *execution history* [1] to make it available to future steps of the analysis. The execution history, $\mathcal{EH}$, is a list of *instruction instances* $\langle s_1, s_2, \ldots, s_n \rangle$. We refer to an instruction instance as $s = i^k$, where i is a program instruction and the superscript k distinguishes between multiple occurrences

Input: j, a conditional jump instruction instance
foreach $f \in use(Rhs(j))$ **do**
  $Replace(Rhs(j), f, Rhs(drd(f, j)))$
  foreach $v \in use(Rhs(j))$ **do**
    if $IsTemporary(v)$ **then**
      $Replace(Rhs(j), v, Rhs(drd(v, j)))$

$Replace(Rhs(j), Rhs(j), Simplify(Rhs(j)))$

Figure 3: Algorithm for *RecoverPredicate*.

of the same instruction in the execution history. When an instruction is executed, the execution history is updated accordingly. By keeping an execution history, we are able, at any program point and for every program variable, to precisely compute the *dynamic reaching definitions* and then directly formulate queries about the data-flow.

For example, after the execution of the code in Figure 1 with input `buf == "a'"`, the execution history is $\mathcal{EH} = \langle 5^1, 6^1, 7^1, 8^1, 9^1, 6^2, 7^2, 8^2, 9^2, 10^1, 6^3, 14^1, 16^1 \rangle$.

The *dynamic reaching definition* [1] $drd(v, s_k)$ of a variable v used by instruction instance s_k is an instruction instance s_{k-j} such that $v \in define(s_{k-j})$ and $\nexists\, s_m, m \in (k-j, k) : v \in define(s_m)$.

Let us now suppose we are interested in computing the dynamic reaching definition for the variable j used by instruction instance 14^1. We can simply scan the execution history backwards, starting from entry 14^1 and searching for the first instruction instance that defines j. Therefore the dynamic reaching definition for j, when used by the instruction instance 14^1, is $drd(j, 14^1) = 10^1$.

4.2.2 Path-conditions

Whenever a conditional control transfer instruction is executed by the program, we perform the path-conditions analysis. The main problem at this point is that the predicates of conditional control transfer instructions are expressed only through control flags which store the result of the computation representing the evaluation of the predicates. In order to generate meaningful path conditions we have to extrapolate the semantic of predicates with respect to input data.

As an example, the assembly fragment in Figure 2 shows how the high-level predicate `buf[i] >= 32` from line 7 of the program in Figure 1 has been translated by the compiler. Note that the condition evaluated by the `jle` has been negated by the compiler, i.e., if the condition is true the "then" block will be skipped. Moreover, the conditional jump statement implicitly uses `ZF`, `OF` and `SF` control flags: these side effects are made explicit by the intermediate representation.

The algorithm we have developed to translate low-level conditional control transfer instructions into high-level predicates is reported in Figure 3. It relies on the following definitions:

- $Simplify(e)$, returns the expression resulting from the simplification of the expression e;
- $Replace(i, e, e')$, substitutes all occurrences of the expression e with the expression e' in i;
- $Rhs(i)$, returns the right-hand side of instruction i; in conditional jumps this corresponds to the branch condition.

The algorithm is applied on a conditional jump instruction instance and works as follows. All occurrences of control flags in the expression representing the jump predicate are replaced with the dynamic reaching definitions (the propagation may be applied twice to avoid temporary variables) and the resulting expression is then transformed and simplified by our simplification engine which is able to manipulate logical and arithmetic terms to express them in a simpler and canonical form. Eventually, the condition of an instruction like `jle 8` will be translated in a form as `m8[r32(EAX)]` $\leq$ `c8(0x1f)`, meaning that the content of the 8-bit memory region with address `EAX` must be less than or equal to the 8-bit constant `0x1f`.

Jump conditions are then translated into *path conditions over the input data*. The algorithm we use to perform this step is described in Figure 4. The output is a jump instruction instance with a conditional expression whose terms are either constants or memory regions related to the input.

It is worth noting that in general the search for a dynamic reaching definition may be complicated by the overlapping of memory locations. In fact, in the worst case, up to n different assignment statements, each one defining a single different byte inside the memory range $[m, m + (n-1)]$, could contribute to the dynamic reaching definition of a n bytes long memory region with base address m. In the algorithm described by Figure 4, $\overline{drd}$ takes care of these problems, thus $r = \overline{drd}(v, s_k)$ returns a virtual instruction that represents the combination of every program assignment instruction that contributes to the definition of variable v at a particular execution step.

The condition of the previous example statement (`jle 8`), became `m8[r32(EAX)]` $\leq$ `c8(0x1f)` after intermediate translation, predicate recovery and simplification, should be now the target of the recursive reaching definitions propagation, which gives the final form of `m8[c32(0xbffffae4)]` $\leq$ `c8(0x1f)`. The address `0xbffffae4` turns out to be a location belonging to what in the source code is called `buf`. Moreover, supposing that the variable `buf` is directly controlled by some input data, no more propagation is needed and the path condition `buf[i] <= 31` is correctly deduced by the dynamic analysis.

Input: j, a conditional jump or assignment instruction instance; $\mathcal{I}$, the set of input variables

foreach $v \in use(Rhs(j))$ **do**
  if $v \in \mathcal{I}$ **then**
    do nothing as the expression already represents the input
  else if $\nexists\, r : r = \overline{drd}(v, j)$ **then**
    $Replace(Rhs(j), v, \mathcal{PSS}[v])$
  else
    $Replace(Rhs(j), v, Rhs(PropagateDRD(\overline{drd}(v, j))))$

$Replace(Rhs(j), Rhs(j), Simplify(Rhs(j)))$
return j

Figure 4: Algorithm for $PropagateDRD$.

Several conditions have to hold to reach a given program point, thus we join the new condition to the existing path conditions. As an example consider the program in Figure 1 with the input buffer `buf == "a'"`. Before executing for the first time the assignment at line 8, the path condition[3] is $\mathcal{PC} = \{buf_0 \neq 0 \wedge buf_1 \neq 0 \wedge buf_2 = 0 \wedge buf_0 > 31 \wedge buf_0 \neq 127\}$[4].

Recurring patterns in conditions are translated into higher-level concepts such as "*length of a string*" or "*(dis)equality of two memory areas*". For example, the above condition is equivalent to asserting that the `buf` string must be 3 bytes long: $\mathcal{PC} = \{strlen(buf) = 2 \wedge buf_0 > 31 \wedge buf_0 \neq 127\}$. Although in our example `strlen` was invoked directly, the high-level condition was automatically generated during the analysis by tracing the execution of the library function. Thus, such condition can be reconstructed not only when `strlen` code is inlined by the compiler or linked statically into the executable, but also when it is reimplemented from scratch.

4.2.3 Loop Analysis

As far as dynamic analysis is concerned, loops are problematic, since in principle they should be executed any possible number of times. In fact, in general, variable definitions can be different for any different iteration. We perform a preliminary loop analysis to try to reduce the relevant iterations by estimating the loop conditions that lead to modifications of sensitive memory locations. While loop identification could be performed off-line, loop conditions and loop induction variables[5] are approximated dynamically.

[3] For clarity memory addresses are represented symbolically by buf_i.

[4] The test $buf_0 \neq 127$ is due to the translation of the source-level inequality `buf[i] < 127`: `GCC` allocates a `char` variable as a 1-byte *signed* memory location, ranging in $[-128, 127]$.

[5] An induction variable is defined as a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable.

With our loop analysis, we are able to perform some kind of abstraction of the loop's behavior from the particular input data set related to current execution. By observing the relationship that exists between loop condition and input dependent data and how a loop induction variable could affect the destination address of a memory assignment, we can define a virtually harmful set of input constraints without actually trying every possible number of loop iterations.

4.3 Heuristics

Heuristic rules are needed to identify sensitive memory regions and execution paths that could lead to security violations. No completeness guarantee can be provided, but our strategy aims at concentrating fuzzing efforts in likely critical paths.

4.3.1 Sensitive Memory Corruption Detection

During runtime analysis, each data that originates or is derived from an untrusted source is marked as *tainted*: we start by marking input data as tainted, and then we dynamically keep track of how the tainted attribute propagates to other data.

Similarly, we start with a minimal set of sensitive areas (e.g., `main()` return address is immediately marked as sensitive). As the execution goes on, new sensitive memory regions are found or existing ones are discarded. As an example, when a new procedure is executed the memory region that holds the return address pushed on the stack by the `call` instruction is marked as sensitive until that procedure returns. Targets of indirect control transfer instructions and data structures employed for dynamically allocating memory regions (i.e., maintenance heap data structures used by `malloc`–like C library functions) are considered sensitive, since they constitute common targets for *heap overflow* attacks [14]. Even the entire `.got` and `.dtors` sections could be marked as sensitive.

To detect when sensitive data could be overwritten or when the execution flow could be maliciously hijacked, during runtime analysis we search for indirect control transfer statements whose target address is tainted, or for intermediate assignment instructions like `*addr = d` where: (I) the address `addr` of the memory location defined by the instruction is a tainted, or (II) `addr` is the address of a sensitive memory location and `d` is a tainted memory region.

Moreover, we associate the corruption of other data, that we can not classify as sensitive (e.g. local user-defined variables), to segmentation faults raised during the execution.

4.3.2 Critical execution paths

Typical rules that can be used to select the paths that should be considered critical to cover during fuzzing are the following:

1. for every conditional jump statement always try to visit a still unexplored code region.
2. Explore paths that involve memory assignments with a non-constant target address and avoid paths that don't.
3. When the loop analysis (see Section 4.2.3) fails to reduce the number of iterations, try to push a loop to perform more iterations; if the loop doesn't seem to write towards sensitive memory regions, as a first approximation, give up with that loop.
4. Avoid *dead-ends*, i.e. paths that terminate with an `exit`-like system call and free from "interesting" statements (i.e. memory assignments with non-constant target address, function calls, ...).
5. When statically one or more reaching definitions for the target address of an indirect control transfer (i.e. `call *%eax`) are identified, each path that contains those assignments should be forced to be visited.

5 Related Works

Fuzzing [16] is a software testing technique that consist in feeding an application with *random* input data. If the program fails (e.g., by crashing) then a bug has been found. Despite recently many protocol-dependent fuzzers have appeared, the original *fuzz* testing uses simple black-box random input: no knowledge of the application is used in generating the random input. Nevertheless, the amount of software faults found with this approach can be impressive [15], even if program bugs found with such method are very simple ones.

A significant part of recent efforts in vulnerability analysis have been directed towards the analysis of software written in high-level languages. An important result in this field is described in [5]: EXE is a dynamic analysis tool able to generate inputs that crash real code, by running it on symbolic input [11] and tracking the constraints on each symbolic memory location. An important difference between this approach and ours is that EXE works on C source code and requires that programmers mark which memory locations should be treated as holding symbolic data. The program is then recompiled with the EXE compiler, that adds instrumentation code to perform symbolic execution. The major drawback of this approach is the need of the source code. Moreover, EXE can fail to identify certain bugs and vulnerabilities because of the mismatch that exists between what the programmers intend and what is actually executed by the processor: security vulnerabilities can exist because of platform-specific details such as memory layout details, register usage, execution order, optimizations performed by the compiler and even compiler bugs [3].

The work described in [7] is closely related to ours. It describes an approach to the identification of vulnerabilities in x86 executables in ELF binary format, based on static analysis and symbolic execution techniques. Their approach uses the information extracted during symbolic execution to statically detect when not sanitized untrusted data could be used as parameter of `system()` and `popen()` standard C library functions. This method differs from ours in a number of ways: first, their work is focused on the detection of potentially harmful uses of not sanitized untrusted data as parameter of dangerous system calls; second, performing only static analysis on the executable forces to accept a rather high false positive rate and, on the other side, to take some optimistic assumptions on several static analysis problems that can even result in missing a potentially significant number of vulnerabilities (e.g., the authors assume that no aliasing occurs and limit loop analysis to three iterations).

In [10], the authors describe a *dynamic taint analysis* method for the automatic detection, analysis and signature generation of exploits on commodity software. They label data originating or derived from untrusted sources as tainted and then keep track of the propagation of tainted data as the program executes, detecting when such data is used in dangerous ways. The authors have implemented TaintCheck, a prototype that performs dynamic taint analysis running the program in its own emulation environment, using the Valgrind framework [17]. It tries to identify an attack while it is in progress and to automatically provide information about how the vulnerability was exploited and which part of the payload led to the exploit of the vulnerability.

Dynamic taint analysis has also been used in [21] to prevent a broader class of attacks (e.g., command injection) than memory corruption by enforcing security policies on tainted data. Although the analysis is performed on source code, the idea of policies on tainted data can be used also in our context to implement a finer grained detector for the identification of more security flaws.

The application of program analysis techniques to machine code is a very active research area, not limited to vulnerability analysis. As an example, in [12, 8] the authors describe how symbolic execution techniques can be employed to detect patterns of malware behavior in binary code, while in [13] static analysis is applied to kernel modules in order to detect kernel-level rootkits.

6 Conclusions and Future Works

We have presented the design of a "smart fuzzer" for the automatic identification of security-relevant flaws in stripped dynamically-linked x86 binary executables, based on a hybrid static/dynamic program analysis. We have implemented a prototype tool that works on Executable and Linking Format (ELF) binaries. Although additional parsing and disassembling would be required to process different object formats and different instruction sets, the technique itself can be easily ported to other platforms.

Currently our prototype is able to perform the static and dynamic analyses described in Section 4, generating the path conditions associated to a particular execution flow. However, these conditions can actually be resolved only invoking a constraint solver manually, thus the iterative analysis has to be performed manually. We are working on (I) the integration of an off-the-shelf constraint solver to overcome such limitation, (II) improving loop analysis to deal with both reducible and irreducible loops and (III) reducing the overhead of on-line analysis, mostly due to the use of single-stepping, to be able to test our framework with real applications.

7 Acknowledgments

We would like to thank the anonymous reviewers for their useful suggestions and comments on this paper.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic Slicing in the Presence of Unconstrained Pointers. In *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 60–73, New York, NY, USA, 1991. ACM Press.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, Zurich, Switzerland, October 2005.

[4] R. Banfi, D. Bruschi, and E. Rosti. Oboe: Object-code buffer overflow evaluator. In *Proc. of the European Symposium on Research in Computer Security, ESORICS '98*, volume LNCS 1485, pages 17–31, Louvain la Neuve (Belgium), Sept. 1998. Springer Verlag.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[6] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software—Practice and Experience*, 25(7):811–829, July 1995.

[7] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, December 2006.

[8] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. *Operating Systems Review*, 40(5):25–36, Dec. 2006.

[9] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, May 9, 2003.

[10] D. S. James Newsome. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), July 1976.

[12] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[13] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 91–100, Tucson, AZ, December 2004.

[14] Michel Kaempf. Smashing The Heap For Fun And Profit. *Phrack Magazine*, 11(57), 2001.

[15] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, University of Wisconsin-Madison, April 1995.

[16] B. P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Comm. of the ACM*, 33(12):32, December 1990.

[17] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV '03)*, Boulder, Colorado, USA, July 2003.

[18] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of Executable Code Revisited. In *9th Working Conference on Reverse Engineering (WCRE)*, pages 45–54, Richmond, VA, USA, 2002. IEEE Computer Society.

[19] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4), 2006.

[20] Wikipedia. Corner case — wikipedia, the free encyclopedia, 2006. [Online; accessed 22-January-2007].

[21] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.