Automated Whitebox Fuzz Testing

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Michael Y. Levin
Microsoft Center for Software Excellence
mlevin@microsoft.com

David Molnar
UC Berkeley
(Visiting Microsoft)
t-davidm@microsoft.com
dmolnar@eecs.berkeley.edu

ABSTRACT

Fuzz testing is an effective technique for finding security vulnerabilities in software. Traditionally, fuzz testing tools apply random mutations to well-formed inputs and test the program on the resulting values. We present an alternative *whitebox fuzz testing* approach inspired by recent advances in symbolic execution and dynamic test generation. Our approach records an actual run of a program under test on a well-formed input, symbolically evaluates the recorded trace, and generates constraints capturing how the program uses its inputs. The generated constraints are used to produce new inputs which cause the program to follow different control paths. This process is repeated with the help of a code-coverage maximizing heuristic designed to find defects as fast as possible. We have implemented this algorithm in SAGE (*Scalable, Automated, Guided Execution*), a new tool employing x86 instruction-level tracing and emulation for whitebox fuzzing of arbitrary file-reading Windows applications. We describe key optimizations needed to make dynamic test generation scale to large input files and long execution traces with hundreds of millions of instructions. We then present detailed experiments with several Windows applications. Notably, without any format-specific knowledge, SAGE detects the MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis tools. Furthermore, while still in an early stage of development, SAGE has already discovered 20+ new bugs in large shipped Windows applications including image processors, media players, and file decoders. Several of these bugs are potentially exploitable memory access violations.

1. INTRODUCTION

Since the "Month of Browser Bugs" released a new bug each day of July 2006 [23], *fuzz testing* has leapt to prominence as a quick and cost-effective method for finding serious security defects in large applications. Fuzz testing is a form of *blackbox random* testing which randomly mutates well-formed inputs and tests the program on the resulting data [11, 28, 1, 4]. In some cases, *grammars* are used to generate the well-formed inputs, which also allows encoding application-specific knowledge and test heuristics. Although fuzz testing can be remarkably effective, the limitations of blackbox testing approaches are well-known. For instance, the `then` branch of the conditional statement "`if (x==10) then`" has only one in 2^{32} chances of being exercised if `x` is a randomly chosen 32-bit input value. This intuitively explains why random testing usually provides low code coverage [26].

We propose a conceptually simple but different approach of *whitebox fuzz testing*. This work is inspired by recent advances in systematic dynamic test generation [14, 7]. Starting with a fixed input, our algorithm symbolically executes the program, gathering input constraints from conditional statements encountered along the way. The collected constraints are then systematically negated and solved with a constraint solver, yielding new inputs that exercise different execution paths in the program. This process is repeated using a novel search algorithm with a coverage-maximizing heuristic designed to find defects as fast as possible. For example, symbolic execution of the above fragment on the input $\mathtt{x} = \mathtt{0}$ generates the constraint $\mathtt{x} \neq \mathtt{10}$. Once this constraint is negated and solved, it yields $\mathtt{x} = \mathtt{10}$, which gives us a new input that causes the program to follow the `then` branch of the given conditional statement.

In theory, systematic dynamic test generation can lead to full program path coverage, i.e., program verification [14]. In practice, however, the search is typically incomplete both because the number of execution paths in the program under test is huge and because symbolic execution, constraint generation, and constraint solving are necessarily imprecise. (See Section 2 for various reasons of why the latter is the case.) Therefore, we are forced to explore practical tradeoffs, and this paper presents what we believe is a particular *sweet spot*. Indeed, our specific approach has been remarkably effective in finding new defects in large applications that were previously well-tested. In fact, our algorithm finds so many defect occurrences that we must address the *defect triage problem* (see Section 4), which is common in static program analysis and blackbox fuzzing, but has not been faced until now in the context of dynamic test generation [14, 7, 29, 22, 20, 16]. Another novelty of our approach is that we test larger applications than previously done in dynamic test generation [14, 7, 29].

We have implemented this approach in SAGE, short for *Scalable, Automated, Guided Execution*, a whole-program whitebox file fuzzing tool for x86 Windows applications. As

Technical Report. Please contact the authors to check for an updated version before citing or redistributing.

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Figure 1: Example of program.

argued above, SAGE is capable of finding bugs that are beyond the reach of blackbox fuzzers. For instance, without any format-specific knowledge, SAGE detects the MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing. Our work makes three main contributions:

- Section 2 introduces a new search algorithm for systematic test generation that is optimized for large applications with large input files and exhibiting long execution traces where the search is bound to be incomplete;
- Section 3 discusses the implementation of SAGE: the engineering choices behind its symbolic execution algorithm and the key optimization techniques enabling it to scale to program traces with hundreds of millions of instructions;
- Section 4 describes our experience with SAGE: we give examples of discovered defects and discuss the results of various experiments.

2. A WHITEBOX FUZZING ALGORITHM

2.1 Background: Dynamic Test Generation

Consider the program shown in Figure 1. This program takes 4 bytes as input and contains an error when the value of the variable `cnt` is greater than or equal to `3` at the end of the function `top`. Running the program with random values for the 4 input bytes is unlikely to discover the error: there are 5 values leading to the error out of $2^{(8*4)}$ possible values for 4 bytes, i.e., a probability of about $1/2^{30}$ to hit the error with random testing, including blackbox fuzzing. This problem is typical of random testing: it is difficult to generate input values that will drive the program through all its possible execution paths.

In contrast, whitebox *dynamic test generation* can easily find the error in this program: it consists of executing the program starting with some initial inputs, performing a dynamic symbolic execution to collect constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. This process is repeated until a given specific program statement or path is executed [20, 16], or until all (or many) feasible program paths of the program are exercised [14, 7].

For the example above, assume we start running the function `top` with the initial 4-letters string `good`. Figure 2 shows the set of all feasible program paths for the function `top`. The leftmost path represents the first run of the program on input `good` and corresponds to the program path ρ including all 4 else-branches of all conditional if-statements in the program. The leaf for that path is labeled with `0` to denote the

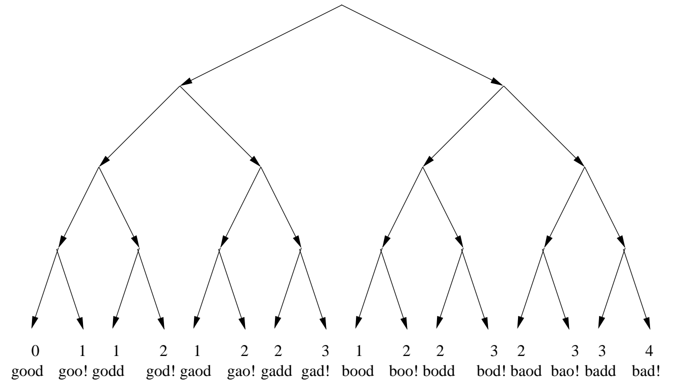

Figure 2: Search space for the example with the value of the variable `cnt` at the end of each run.

value of the variable `cnt` at the end of the run. Intertwined with the normal execution, a symbolic execution collects the predicates $i_0 \neq \texttt{b}$, $i_1 \neq \texttt{a}$, $i_2 \neq \texttt{d}$ and $i_3 \neq \texttt{!}$ according to how the conditionals evaluate, and where i_0, i_1, i_2 and i_3 are *symbolic variables* that represent the values of the memory locations of the input variables `input[0]`, `input[1]`, `input[2]` and `input[3]`, respectively.

The path constraint $\phi_\rho = \langle i_0 \neq \texttt{b}, i_1 \neq \texttt{a}, i_2 \neq \texttt{d}, i_3 \neq \texttt{!} \rangle$ represents an equivalence class of input vectors, namely all the input vectors that drive the program through the path that was just executed. To force the program through a different equivalence class, one can calculate a solution to a different path constraint, say, $\langle i_0 \neq \texttt{b}, i_1 \neq \texttt{a}, i_2 \neq \texttt{d}, i_3 = \texttt{!} \rangle$ obtained by negating the last predicate of the current path constraint. A solution to this path constraint is $(i_0 = \texttt{g}, i_1 = \texttt{o}, i_2 = \texttt{o}, i_3 = \texttt{!})$. Running the program `top` with this new input `goo!` exercises a new program path depicted by the second leftmost path in Figure 2. By repeating this process, the set of all 16 possible execution paths of this program can be exercised. If this systematic search is performed in depth-first order, these 16 executions are explored from left to right on the Figure. The error is then reached for the first time with `cnt==3` during the 8th run, and full branch/block coverage is achieved after the 9th run.

2.2 Limitations

Systematic dynamic test generation [14, 7] as briefly described above has two main limitations:

Path explosion: systematically executing all feasible program paths does not scale to large, realistic programs. Path explosion can be alleviated by performing dynamic test generation *compositionally* [12], by testing functions in isolation, encoding test results as *function summaries* expressed using function input preconditions and output postconditions, and then re-using those summaries when testing higher-level functions. Although the use of summaries in software testing seems promising, achieving full path coverage when testing large applications with hundreds of millions of instructions is still problematic within a limited search period, say, one night, even when using summaries.

Imperfect symbolic execution: symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating-system and library functions that are hard or impossible to reason about sym-

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) { // new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

Figure 3: Search algorithm.

bolically with good enough precision at a reasonable cost. Whenever symbolic execution is not possible, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution [14]. Randomization can also help by suggesting concrete values whenever automated reasoning is difficult. Whenever an actual execution path does not match the program path predicted by symbolic execution for a given input vector, we say that a *divergence* has occurred. Divergences can be detected by recording predicted execution paths as bit vectors (one bit for each conditional branch outcome) and checking that that the expected path is actually taken in subsequent test runs.

2.3 Generational Search

We now present a new search algorithm that is designed to address these fundamental practical limitations. Specifically, our algorithm has the following prominent features:

- it is designed to systematically yet partially explore the state spaces of large applications executed with large inputs (thousands of symbolic variables) and with very deep paths (hundreds of millions of instructions);
- it maximizes the number of new tests generated from each symbolic execution (which are long and expensive in our context) while avoiding any redundancy in the search;
- it uses heuristics to maximize code (block) coverage as quickly as possible, with the goal of finding bugs faster;
- it is resilient to divergences: whenever divergences occur, the search is able to recover and continue.

This new search algorithm is presented in two parts in Figures 3 and 4. The main `Search` procedure of Figure 3 is mostly standard. It places the initial input `inputSeed` in a `workList` (line 3) and runs the program to check whether any bugs are detected during the first execution (line 4). The inputs in the `workList` are then processed (line 5) by selecting an element (line 6) and expanding it (line 7) to generate new inputs with the function `ExpandExecution` described later in Figure 4. For each of those `childInputs`, the program under test is run with that input. This execution is checked for errors (line 10) and is assigned a `Score` (line 11), as discussed below, before being added to the `workList` (line 12) which is sorted by those scores.

```
1 ExpandExecution(input) {
2   childInputs = {};
3   // symbolically execute (program,input)
4   PC = ComputePathConstraint(input);
5   for (j=input.bound; j < |PC|; j++) {
6     if((PC[0..(j-1)] and not(PC[j])) has a solution I){
7       newInput = input + I;
8       newInput.bound = j;
9       childInputs = childInputs + newInput;
10  }
11  return childInputs;
12}
```

Figure 4: Computing new children.

The originality of our search algorithm is in the way children are expanded as described in Figure 4. Given an `input` (line 1), the function `ExpandExecution` symbolically executes the program under test with that `input` and generates a *path constraint* `PC` (line 4) as defined earlier. `PC` is a conjunction of `|PC|` constraints, each corresponding to a conditional statement in the program and expressed using symbolic variables representing values of input parameters (see [14, 7]). Then, our algorithm attempts to *expand every* constraint in the path constraint (at a position `j` greater or equal to a parameter called `input.bound` which is initially `0`). This is done by checking whether the conjunction of the part of the path constraint prior to the `j`th constraint `PC[0..(j-1)]` and of the negation of the `j`th constraint `not(PC[j])` is satisfiable. If so, a solution `I` to this new path constraint is used to update the previous solution `input` while values of input parameters not involved in the path constraint are preserved (this update is denoted by `input + I` on line 7). The resulting new input value is saved for future evaluation (line 9).

In other words, starting with an initial input `inputSeed` and initial path constraint `PC`, the new search algorithm depicted in Figures 3 and 4 will attempt to expand *all* `|PC|` constraints in `PC`, instead of just one, such as the last one with a *depth-first search*, or the first one with a *breadth-first search*. To prevent these child sub-searches from redundantly exploring overlapping parts of the search space, a parameter `bound` is used to limit the backtracking of each sub-search above the branch where the sub-search started off its parent. Because each execution is typically expanded with many children, we call such a search order a *generational search*.

Consider again the program shown in Figure 1. Assuming the initial input is the 4-letters string `good`, the leftmost path in the tree of Figure 2 represents the first run of the program on that input. From this *parent* run, a generational search generates four *first-generation* children which correspond to the four paths whose leafs are labeled with `1`. Indeed, those four paths each correspond to negating *one* constraint in the original path constraint of the leftmost parent run. Each of those first generation execution paths can in turn be expanded by the procedure of Figure 4 to generate (zero or more) *second-generation* children. There are six of those and each one is depicted with a leaf label of `2` to the right of their (first-generation) parent in Figure 2. By repeating this process, all feasible execution paths of the function `top` are eventually generated exactly once. For this example, the value of the variable `cnt` denotes exactly the

generation number of each run.

Since the procedure `ExpandExecution` of Figure 4 expands all constraints in the current path constraint (below the current `bound`) instead of just one, it maximizes the number of new test inputs generated from each symbolic execution. Although this optimization is perhaps not significant when exhaustively exploring all execution paths of small programs like the one of Figure 1, it is important when *symbolic execution takes a long time*, as is the case for large applications where exercising all execution paths is virtually hopeless anyway. This point will be further discussed in Section 3 and illustrated with the experiments reported in Section 4.

In this scenario, we want to exploit as much as possible the first symbolic execution performed with an initial input and to systematically explore all its first-generation children. This search strategy works best if that initial input is *well formed*. Indeed, it will be more likely to exercise more of the program's code and hence generate more constraints to be negated, thus more children, as will be shown with experiments in Section 4. The importance given to the first input is similar to what is done with traditional, blackbox fuzz testing, hence our use of the term *whitebox fuzzing* for the search technique introduced in this paper.

The expansion of the children of the first parent run is itself prioritized by using a heuristic to attempt to maximize block coverage as quickly as possible, with the hope of finding more bugs faster. The function `Score` (line 11 of Figure 3) computes the incremental block coverage obtained by executing the `newInput` compared to all previous runs. For instance, a `newInput` that triggers an execution uncovering 100 new blocks would be assigned a score of 100. Next, (line 12), the `newInput` is inserted into the `workList` according to its score, with the highest scores placed at the head of the list. Note that all children compete with each other to be expanded next, regardless of their generation number.

Our block-coverage heuristic is related to the "Best-First Search" of EXE [7]. However, the overall search strategy is different: while EXE uses a depth-first search that occasionally picks the next child to explore using a block-coverage heuristic, a generational search tests all children of each expanded execution, and scores their entire runs before picking the best one from the resulting `workList`.

The block-coverage heuristics computed with the function `Score` also helps dealing with *divergences* as defined in the previous section, i.e., executions diverging from the expected path constraint to be taken next. The occurrence of a single divergence compromises the completeness of the search, but this is not the main issue in practice since the search is bound to be incomplete for very large search spaces anyway. A more worrisome issue is that divergences may prevent the search from making any progress. For instance, a depth-first search which diverges from a path p to a previously explored path p' would cycle forever between that path p' and the subsequent divergent run p. In contrast, our generational search tolerates divergences and can recover from this pathological case. Indeed, each run spawns many children, instead of a single one as with a depth-first search, and, if a child run p divergences to a previous one p', that child p will have a zero score and hence be placed at the end of the `workList` without hampering the expansion of other, non-divergent children. Dealing with divergences is another important feature of our algorithm for handling large applications for which symbolic execution is bound to be imperfect/incomplete, as will be demonstrated in Section 4.

Finally, we note that the generational search parallelizes well, since children can be checked and scored independently; only the work list and overall block coverage need to be shared.

3. THE SAGE SYSTEM

The generational search algorithm presented in the previous section has been implemented in a new tool named SAGE, which stands for *Scalable, Automated, Guided Execution*. SAGE can test any file-reading program running on Windows by treating bytes read from files as symbolic inputs. Another key novelty of SAGE is that it performs symbolic execution of program traces at the x86 binary level. This section justifies this design choice by arguing how it allows SAGE to handle a wide variety of large production applications. This design decision raises challenges that are different from those faced by source-code level symbolic execution. We describe these challenges and show how they are addressed in our implementation. Finally, we outline key optimizations that are crucial in scaling to large programs.

3.1 System Architecture

SAGE performs a generational search by repeating four different types of tasks. The `Tester` task implements the function `Run&Check` by executing a program under test on a test input and looking for unusual events such as access violation exceptions and extreme memory consumption. The subsequent tasks proceed only if the `Tester` task did not encounter any such errors. If `Tester` detects an error, it saves the test case and performs automated triage as discussed in Section 4.

The `Tracer` task runs the target program on the same input file again, this time recording a log of the run which will be used by the following tasks to replay the program execution offline. This task uses the `iDNA` framework [3] to collect complete execution traces at the machine-instruction level.

The `CoverageCollector` task replays the recorded execution to compute which basic blocks were executed during the run. SAGE uses this information to implement the function `Score` discussed in the previous section.

Lastly, the `SymbolicExecutor` task implements the function `ExpandExecution` of Section 2.3 by replaying the recorded execution once again, this time to collect input-related constraints and generate new inputs using the constraint solver `Disolver` [17].

Both the `CoverageCollector` and `SymbolicExecutor` tasks are built on top of the trace replay framework `TruScan` [24] which consumes trace files generated by `iDNA` and virtually re-executes the recorded runs. `TruScan` offers several features that substantially simplify symbolic execution. These include instruction decoding, providing an interface to program symbol information, monitoring various input/output system calls, keeping track of heap and stack frame allocations, and tracking the flow of data through the program structures.

3.2 Trace-based x86 Constraint Generation

SAGE's constraint generation differs from previous dynamic test generation implementations [14, 29, 7] in two main ways. First, instead of a source-based instrumentation, SAGE adopts a *machine-code-based* approach for three

main reasons:
Multitude of languages and build processes. Source-based instrumentation must support the specific language, compiler, and build process for the program under test. There is a large upfront cost for adapting the instrumentation to a new language, compiler, or build tool. Covering many applications developed in a large company with a variety of incompatible build processes and compiler versions is a logistical nightmare. In contrast, a machine-code based symbolic-execution engine, while complicated, need be implemented only once per architecture. As we will see in Section 4, this choice has let us apply SAGE to a large spectrum of production software applications.
Compiler and post-build transformations. By performing symbolic execution on the binary code that actually ships, SAGE makes it possible to catch bugs not only in the target program but also in the compilation and post-processing tools, such as code obfuscators and basic block transformers, that may introduce subtle differences between the semantics of the source and the final product.
Unavailability of source. It might be difficult to obtain source code of third-party components, or even components from different groups of the same organization. Source-based instrumentation may also be difficult for self-modifying or JITed code. SAGE avoids these issues by working at the machine-code level. While source code does have information about types and structure not immediately visible at the machine code level, we do not need this information for SAGE's path exploration.

Second, instead of an online instrumentation, SAGE adopts an *offline trace-based* constraint generation. With online generation, constraints are generated as the program is executed either by statically injected instrumentation code or with the help of dynamic binary instrumentation tools such as Nirvana [3] or `Valgrind` [25] (`Catchconv` is an example of the latter approach [22].) SAGE adopts offline trace-based constraint generation for two reasons. First, a single program may involve a large number of binary components some of which may be protected by the operating system or obfuscated, making it hard to replace them with instrumented versions. Second, inherent nondeterminism in large target programs makes debugging online constraint generation difficult. If something goes wrong in the constraint generation engine, we are unlikely to reproduce the environment leading to the problem. In contrast, constraint generation in SAGE is completely deterministic because it works from the execution trace that captures the outcome of all nondeterministic events encountered during the recorded run.

3.3 Constraint Generation

SAGE maintains the concrete and symbolic state of the program represented by a pair of stores associating locations, memory and registers, with byte-sized values and *symbolic tags* respectively. A symbolic tag represents either an input value or a function of some input value. SAGE supports several kinds of tags: `input`(m) represents the mth byte of the input; c represents a constant; t_1 *op* t_2 denotes the result of some arithmetic or bitwise operation *op* on the values represented by the tags t_1 and t_2; the sequence tag $\langle t_0 \dots t_n \rangle$ where $n = 1$ or $n = 3$ describes a word- or double-word-sized value obtained by grouping byte-sized values represented by tags $t_0 \dots t_m$ together; `subtag`(t, i) where $i \in \{0 \dots 3\}$ corresponds to the i-th byte in the word- or double-word-sized value represented by t. Note that SAGE does not reason about symbolic dereference of pointers. SAGE defines a fresh symbolic variable for each non-constant symbolic tag. Provided there is no confusion, we do not distinguish a tag from its associated symbolic variable in the rest of this section.

As SAGE replays the recorded program trace, it updates the concrete and symbolic stores according to the semantics of each visited instruction.

In addition to performing symbolic tag propagation, SAGE also generates *constraints* on input values. Constraints are relations over *symbolic variables*; for example, given a variable x that corresponds to the tag `input`(5), the constraint $x < 10$ denotes the fact that the fifth byte of the input is less than 10.

When the algorithm encounters an input-dependent conditional jump, it creates a constraint modeling the outcome of the branch and adds it to the path constraint composed of the constraints encountered so far.

The following simple example illustrates the process of tracking symbolic tags and collecting constraints.

```
# read 10 byte file into a
# buffer beginning at address 1000
mov ebx, 1005
mov al, byte [ebx]
dec al               # Decrement al
jz LabelForIfZero    # Jump if al == 0
```

The beginning of this fragment uses a system call to read a 10 byte file into the memory range starting from address 1000. For brevity, we omit the actual instruction sequence. As a result of replaying these instructions, SAGE updates the symbolic store by associating addresses 1000 ... 1009 with symbolic tags `input`(0) ... `input`(9) respectively. The two `mov` instructions have the effect of loading the fifth input byte into register `al`. After replaying these instructions, SAGE updates the symbolic store with a mapping of `al` to `input`(5). The effect of the last two instructions is to decrement `al` and to make a conditional jump to `LabelForIfZero` if the decremented value is 0. As a result of replaying these instructions, depending on the outcome of the branch, SAGE will add one of two constraints $t = 0$ or $t \neq 0$ where $t =$ `input`$(5) - 1$. The former constraint is added if the branch is taken; the latter if the branch is not taken.

This leads us to one of the key difficulties in generating constraints from a stream of x86 machine instructions—dealing with the two-stage nature of conditional expressions. When a comparison is made, it is not known how it will be used until a conditional jump instruction is executed later. The processor has a special register EFLAGS that packs a collection of status flags such as CF, SF, AF, PF, OF, and ZF. How these flags are set is determined by the outcome of various instructions. For example, CF—the first bit of EFLAGS—is the carry flag that is influenced by various arithmetic operations. In particular, it is set to 1 by a subtraction instruction whose first argument is less than the second. ZF is the zero flag located at the seventh bit of EFLAGS; it is set by a subtraction instruction if its arguments are equal. Complicating matters even further, some instructions such as `sete` and `pushf` access EFLAGS directly.

For sound handling of EFLAGS, SAGE defines bitvector tags of the form $\langle f_0 \dots f_{n-1} \rangle$ describing an n-bit value whose

bits are set according to the constraints $f_0 \dots f_{n-1}$. In the example above, when SAGE replays the `dec` instruction, it updates the symbolic store mapping for `al` and for EFLAGS. The former becomes mapped to $\texttt{input}(5) - 1$; the latter—to the bitvector tag $\langle t < 0 \dots t = 0 \dots \rangle$ where $t = \texttt{input}(5) - 1$ and the two shown constraints are located at offsets 0 and 6 of the bitvector—the offsets corresponding to the positions of CF and ZF in the EFLAGS register.

Another pervasive x86 practice involves casting between byte, word, and double word objects. Even if the main code of the program under test does not contain explicit casts, it will invariably invoke some run-time library function such as `atol`, `malloc`, or `memcpy` that does.

SAGE implements sound handling of casts with the help of subtag and sequence tags. This is illustrated by the following example.

```
mov ch, byte [...]
mov cl, byte [...]
inc cx                  # Increment cx
```

Let us assume that the two `mov` instructions read addresses associated with the symbolic tags t_1 and t_2. After SAGE replays these instructions, it updates the symbolic store with the mappings $\texttt{cl} \mapsto t_1$ and $\texttt{ch} \mapsto t_2$. The next instruction increments `cx`—the 16-bit register containing `cl` and `ch` as the low and high bytes respectively. Right before the increment, the contents of `cx` can be represented by the sequence tag $\langle t_1, t_2 \rangle$. The result of the increment then is the word-sized tag $t = (\langle t_1, t_2 \rangle + 1)$. To finalize the effect of the `inc` instruction, SAGE updates the symbolic store with the byte-sized mappings $\texttt{cl} \mapsto \texttt{subtag}(t, 0)$ and $\texttt{ch} \mapsto \texttt{subtag}(t, 1)$. SAGE encodes the subtag relation by the constraint $x = x' + 256 * x''$ where the word-sized symbolic variable x corresponds to t and the two byte-sized symbolic variables x' and x'' correspond to $\texttt{subtag}(t, 0)$ and $\texttt{subtag}(t, 1)$ respectively.

3.4 Constraint Optimization

SAGE employs a number of optimization techniques whose goal is to improve the speed and memory usage of constraint generation: *tag caching* ensures that structurally equivalent tags are mapped to the same physical object; *unrelated constraint elimination* reduces the size of constraint solver queries by removing the constraints which do not share symbolic variables with the negated constraint; *equivalent constraint elision* skips a constraint if an equivalent one has already been added to the path condition; *flip count limit* establishes the maximum number of times a constraint generated from a particular program instruction can be flipped; *concretization* reduces the symbolic tags involving bitwise and multiplicative operators into their corresponding concrete values.

These optimizations are fairly standard in dynamic test generation. The rest of this section describes *constraint subsumption*, an optimization we found particularly useful for analyzing structured-file parsing applications.

The constraint subsumption optimization keeps track of the constraints generated from a given branch instruction. When a new constraint f is created, SAGE uses a fast syntactic check to see whether f definitely implies or is definitely implied by another constraint generated from the same instruction. If this is the case, the implied constraint is removed from the path constraint.

The subsumption optimization has a critical impact on many programs processing structured files such as various image parsers and media players. For example, in one of the Media 2 searches described in Section 4, we have observed a ten-fold decrease in the number of constraints because of subsumption. Without this optimization, SAGE runs out of memory and overwhelms the constraint solver with a huge number of redundant queries.

Let us look at the details of the constraint subsumption optimization with the help of the following example:

```
1:  mov cl, byte [...]
2:  dec cl               # Decrement cl
3:  ja 2                 # Jump if cl > 0
```

This fragment loads a byte into `cl` and decrements it in a loop until it becomes 0. Assuming that the byte read by the `mov` instruction is mapped to a symbolic tag t_0, the algorithm outlined in Section 3.3 will generate constraints $t_1 > 0, \dots, t_{k-1} > 0$, and $t_k \leq 0$ where k is the value of the loaded byte and $t_{i+1} = t_i - 1$ for $i \in \{1 \dots k\}$. Here, the memory cost is linear in the number of loop iterations because each iteration produces a new constraint and a new symbolic tag.

The subsumption technique allows us to remove the first $k - 2$ constraints because they are implied by the following constraints. We still have to hold on to a linear number of symbolic tags because each one is defined in terms of the preceding tag. To achieve constant space behavior, constraint subsumption must be performed in conjunction with *constant folding* during tag creation: $(t - c) - 1 = t - (c + 1)$. The net effect of the algorithm with constraint subsumption and constant folding on the above fragment is the path constraint with two constraints $t_0 - (k - 1) > 0$ and $t_0 - k \leq 0$.

Another hurdle arises from multi-byte tags. Consider the following loop which is similar to the loop above except that the byte-sized register `cl` is replaced by the word-sized register `cx`.

```
1:  mov cx, word [...]
2:  dec cx               # Decrement cx
3:  ja 2                 # Jump if cx > 0
```

Assuming that the two bytes read by the `mov` instruction are mapped to tags t'_0 and t''_0, this fragment yields constraints $s_1 > 0, \dots, s_{k-1} > 0$, and $s_k \leq 0$ where $s_{i+1} = \langle t'_i, t''_i \rangle - 1$ with $t'_i = \texttt{subtag}(s_i, 0)$ and $t''_i = \texttt{subtag}(s_i, 1)$ for $i \in \{1 \dots k\}$. Constant folding becomes hard because each loop iteration introduces syntactically unique but semantically redundant word-size sequence tags. SAGE solves this with the help of *sequence tag simplification* which rewrites $\langle \texttt{subtag}(t, 0), \texttt{subtag}(t, 1) \rangle$ into t avoiding duplicating equivalent tags and enabling constant folding.

Constraint subsumption, constant folding, and sequence tag simplification are sufficient to guarantee constant space replay of the above fragment generating constraints $\langle t'_0, t''_0 \rangle - (k - 1) > 0$ and $\langle t'_0, t''_0 \rangle - k \leq 0$. More generally, these three simple techniques enable SAGE to effectively fuzz real-world structured-file-parsing applications in which the input-bound loop pattern is pervasive.

4. EXPERIMENTS

We first describe our initial experiences with SAGE, including several bugs found by SAGE that were missed by blackbox fuzzing efforts. Inspired by these experiences, we pursue a more systematic study of SAGE's behavior on two media-parsing applications. In particular, we focus on the importance of the starting input file for the search, the effect of our generational search vs. depth-first search, and the impact of our block coverage heuristic. In some cases, we withold details concerning the exact application tested because the bugs are still in the process of being fixed.

4.1 Initial Experiences

MS07-017. On 3 April 2007, Microsoft released an out of band security patch for code that parses ANI format animated cursors. The vulnerability was originally reported to Microsoft in December 2006 by Alex Sotirov of Determina Security Research, then made public after exploit code appeared in the wild [30]. The Microsoft SDL Policy Weblog states that extensive blackbox fuzz testing of this code failed to uncover the bug, and that existing static analysis tools are not capable of finding the bug without excessive false positives [18]. SAGE, in contrast, synthesizes a new input file exhibiting the bug within hours of starting from a well-formed ANI file.

In more detail, the vulnerability results from an incomplete patch to MS05-006, which also concerned ANI parsing code. The root cause of this bug was a failure to validate a size parameter read from an ANI file. Unfortunately, the patch for MS05-006 is incomplete. Only the length of the *first* `anih` record is checked. If a file has an initial `anih` record of 36 bytes or less, the check is satisfied but then an icon loading function is called on all `anih` records. The length fields of the second and subsequent records are not checked, so any of these records can trigger memory corruption.

Therefore, a test case needs at least two `anih` records to trigger the MS07-017 bug. The SDL Policy Weblog attributes the failure of blackbox fuzz testing to find MS07-017 to the fact that all of the seed files used for blackbox testing had only one `anih` record, and so none of the test cases generated would avoid the MS05-006 patch. While of course one could write a grammar that generates such test cases for blackbox fuzzing, this requires effort and does not generalize beyond the single ANI format.

In contrast, SAGE can generate a crash exhibiting MS07-017 starting from a well-formed ANI file with one `anih` record, despite having no knowledge of the ANI format. Our seed file was picked arbitrarily from a library of well-formed ANI files, and we analyzed a small test driver that called `user32.dll` to parse test case ANI files. The initial test case generated a path condition with 341 branch constraints after parsing 1279939 total instructions over 10072 symbolic input bytes. SAGE then created a crashing ANI at depth 72 after 7 hours 36 minutes in 7706 test cases, using one core of a 2 GHz AMD Opteron 270 dual-core processor running 32-bit Windows Vista with 4 GB of RAM. Figure 5 shows a prefix of our seed file side by side with the crashing SAGE-generated test case. Figure 6 shows further statistics from this test run.

Compressed File Format. We released an alpha version of SAGE to an internal testing team to look for bugs in code that handles a compressed file format. The parsing code for this file format had been extensively tested with blackbox fuzzing tools, but SAGE found two serious new bugs. The first bug was a stack overflow. The second bug was an infinite loop that caused the processing application to consume nearly 100 of the CPU. Both bugs were fixed within a week of filing. Figure 6 shows statistics from a SAGE run on this test code, seeded with a well-formed compressed file. SAGE also uncovered two separate crashes due to read access violations while parsing malformed files of a different format tested by the same team; the corresponding bugs were also fixed within a week of filing.

```
RIFF...ACONLIST       RIFF...ACONB
B...INFOINAM....      B...INFOINAM....
3D Blue Alternat      3D Blue Alternat
e v1.1..IART....      e v1.1..IART....
................      ................
1996..anih$...$.      1996..anih$...$.
................      ................
................      ................
..rate..........      ..rate..........
..........seq ..      ..........seq ..
................      ................
..LIST....framic      ..anih....framic
on.........  ..       on.........  ..
```

Figure 5: On the left, an ASCII rendering of a prefix of the seed ANI file used for our search. On the right, the SAGE-generated crash for MS07-017. Note how the SAGE test case changes the `LIST` to an additional `anih` record on the next-to-last line.

Media File Parsing. We applied SAGE to parsers for four widely used media file formats, which we will refer to as "Media 1," "Media 2," "Media 3," and "Media 4." Through several testing sessions, SAGE discovered crashes in each of these media files that resulted in nine distinct bug reports. For example, SAGE discovered a read violation due to the program copying zero bytes into a buffer and then reading from a non-zero offset. In addition, starting from a seed file of 100 zero bytes, SAGE synthesized a crashing Media 1 test case after 1403 test cases, demonstrating the power of SAGE to infer file structure from code. Figure 6 shows statistics on the size of the SAGE search for each of these parsers, when starting from a well-formed file.

Office 2007 Application. We have used SAGE to successfully synthesize crashing test cases for a large application shipped as part of Office 2007. Over the course of two 10-hour searches seeded with two different well-formed files, SAGE generated 4548 test cases, of which 43 crashed the application. The crashes we have investigated so far are NULL pointer dereference errors, and they show how SAGE can successfully reason about programs on a large scale. Figure 6 shows statistics from the SAGE search on one of the well-formed files.

Image Parsing. We used SAGE to exercise the image parsing code in a media layer included with a variety of other applications. While our initial run did not find crashes, we used an internal tool to scan traces from SAGE-generated test cases and find several uninitialized value use errors. We reported these errors to the testing team, who expanded the result into a reproducible crash. The experience shows that SAGE can uncover serious bugs that do not immediately

Test	# `SymbolicExecutor`	SymExecTime	Init. `|PC|`	# Tests	Mean Depth	Mean # Instr.	Mean Size
`ANI`	808	19099	341	11468	178	2066087	5400
`Media 1`	564	5625	71	6890	73	3409376	65536
`Media 2`	3	3457	3202	1045	1100	271432489	27335
`Media 3`	17	3117	1666	2266	608	54644652	30833
`Media 4`	7	3108	1598	909	883	133685240	22209
Compressed File	47	1495	111	1527	65	480435	634
`OfficeApp`	1	3108	15745	3008	6502	923731248	45064

Figure 6: Statistics from 10-hour searches on seven test applications, each seeded with a well-formed input file. We report the number of `SymbolicExecutor` tasks during the search, the total time spent in all `SymbolicExecutor` tasks in seconds, the number of constraints generated from the seed file, the total number of test cases generated, the mean depth per test case in number of constraints, the mean number of instructions executed after reading the input file, and the mean size of the symbolic input in bytes.

stack hash	wff-1	wff-2	wff-3	wff-4	wff-5	bogus-1
1867196225	×	×	×	×	×	
2031962117	×	×	×	×	×	
612334691		×	×			
1061959981			×	×		
1212954973			×			×
1011628381			×	×		×
842674295				×		
1246509355			×	×		×
1527393075				×		
1277839407					×	
1392730167					×	
1951025690			×			

stack hash	wff-1	wff-3	wff-4	wff-5
790577684	×	×	×	×
825233195	×	×		×
795945252	×	×	×	×
1060863579	×	×	×	×
1043337003			×	
808455977				×
1162567688				×

Figure 7: On top, stack hashes for Media 1. Below, stack hashes for Media 2. SAGE found 12 distinct stack hashes from 357 Media 1 crashing files and 7 distinct stack hashes from 88 Media 2 crashing files.

lead to crashes. For the rest of this paper, however, we focus only on test cases that immediately cause runtime crashes.

4.2 Experiment Setup

Test Plan. We focused on the Media 1 and Media 2 parsers, because they are widely used. We ran a SAGE search for the Media 1 parser with five "well-formed" media files, chosen from a library of test media files. We also tested Media 1 with five "bogus" files : `bogus-1` consisting of 100 zero bytes, `bogus-2` consisting of 800 zero bytes, `bogus-3` consisting of 25600 zero bytes, `bogus-4` consisting of 100 randomly generated bytes, and `bogus-5` consisting of 800 randomly generated bytes. For each of these 10 files, we ran a 10-hour SAGE search seeded with the file to establish a baseline number of crashes found by SAGE. If a task was in progress at the end of 10 hours, we allowed it to finish, leading to search times slightly longer than 10 hours in some cases. For searches that found crashes, we then re-ran the SAGE search for 10 hours, but disabled our block coverage heuristic. We repeated the process for the Media 2 parser with five "well-formed" Media 2 files and the `bogus-1` file.

Each SAGE search used AppVerifier [8], configured to check for heap errors. We then collected crashing test cases, the absolute number of code blocks covered by the seed input, and the number of code blocks added over the course of the search. We performed our experiments on four machines, each with two dual-core AMD Opteron 270 processors running at 2 GHz. During our experiments, however, we used only one core to reduce the effect of nondeterministic task scheduling on the search results. Each machine ran 32-bit Windows Vista, with 4 GB of RAM and a 250 GB hard drive.

Triage. Because a SAGE search can generate many different test cases that exhibit the same bug, we "bucket" crashing files by the *stack hash* of the crash, which includes the address of the faulting instruction. It is possible for the same bug to be reachable by program paths with different stack hashes for the same root cause. Our experiments always report the distinct stack hashes.

Nondeterminism in Coverage Results. As part of our experiments, we measured the absolute number of blocks covered during a test run. We observed that running the same input on the same program can lead to slightly different initial coverage, even on the same machine. We believe this is due to nondeterminism associated with loading and initializing DLLs used by our test applications.

4.3 Results and Observations

The Appendix shows a table of results from our experiments. Here we comment on some general observations. We stress that these observations are from a limited sample size of two applications and should be taken with caution.

Symbolic execution is slow. We measured the total amount of time spent performing symbolic execution during each search. We observed that a single symbolic execution task is many times slower than testing or tracing a program. For example, the mean time for a symbolic execution task in the Media 2 search seeded with `wff-3` was 25 minutes 30 seconds, while testing a Media 2 file took seconds. At the same time, we also observed that only a small portion of the search time was spent performing symbolic execution, because each task generated many test cases; in the Media 2 `wff-3` case, only 25% of the search time was spent in symbolic execution. This shows how generational search effectively leverages the expensive symbolic execution task. This also shows the benefit of separating the `Tester` task from the more expensive `SymbolicExecutor` task.

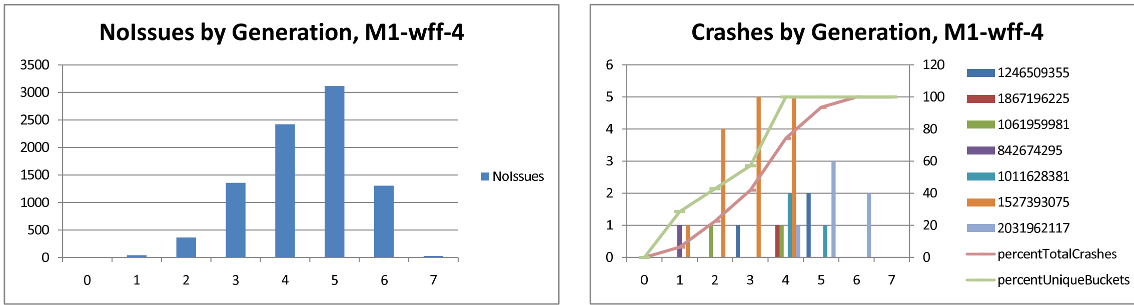

Figure 8: Histograms of test cases and of crashes by generation for Media 1 seeded with `wff-4`.

Generational search is better than depth-first search. We performed several runs with depth-first search. First, we discovered that the SAGE search on Media 1 when seeded with the `bogus-1` file exhibited a pathological divergence (see Section 2) leading to premature termination of the search after 18 minutes. Upon further inspection, this divergence proved to be due to concretizing an AND operator in the path constraint. We did observe depth-first search runs for 10 hours for Media 2 searches seeded with `wff-2` and `wff-3`. Neither depth-first search found crashes. In contrast, while generational search seeded with `wff-2` found no crashes, generational search seeded with `wff-3` found 15 crashing files in 4 buckets. Furthermore, the depth-first searches were inferior to the generational searches in code coverage: the `wff-2` generational search started at 51217 blocks and added 12329, while the depth-first search started with 51476 and added only 398. For `wff-3`, generational search started at 41726 blocks and added 9564, while the depth-first search started at 41703 blocks and added 244. These different start points arise from the nondeterminism noted above, but the difference in blocks added is much larger than the difference in starting coverage. The limitations of depth-first search regarding code coverage are well known (e.g., [21]) and are due to the search being too localized. In contrast, a generational search explores alternative execution branches at all depths, simultaneously exploring all the layers of the program. Finally, we saw that a much larger percentage of the search time is spent in symbolic execution for depth-first search than for generational search, because each test case requires a new symbolic execution task. For example, for the Media 2 search seeded with `wff-3`, 10 hours and 27 minutes were spent in symbolic execution for 18 test cases generated, out of a total of 10 hours and 35 minutes.

Divergences are common. Our basic test setup did not measure divergences, so we ran several instrumented test cases to measure the divergence rate. In these cases, we often observed divergence rates of over 60%. This may be due to several reasons: in our experimental setup, we concretize all non-linear operations (such as multiplication, division, and bitwise arithmetic) for efficiency, there are several x86 instructions we still do not emulate, we do not model symbolic dereference of pointers, tracking symbolic variables may be incomplete, and we do not control all sources of nondeterminism as mentioned above. Despite this, SAGE was able to find many bugs in real applications, showing that our search technique is tolerant of such divergences.

Bogus files find few bugs. We collected crash data from our well-formed and bogus seeded SAGE searches. The bugs found by each seed file are shown, bucketed by stack hash, in Figure 7. Out of the 10 files used as seeds for SAGE searches on Media 1, 6 found at least one crashing test case during the search, and 5 of these 6 seeds were well-formed. Furthermore, all the bugs found in the search seeded with `bogus-1` were also found by at least one well-formed file. For SAGE searches on Media 2, out of the 6 seed files tested, 4 found at least one crashing test case, and all were well-formed. Hence, the conventional wisdom that well-formed files should be used as a starting point for fuzz testing applies to our whitebox approach as well.

Different files find different bugs. Furthermore, we observed that no single well-formed file found all distinct bugs for either Media 1 or Media 2. This suggests that using a wide variety of well-formed files is important for finding distinct bugs.

Bugs found are shallow. For each seed file, we collected the maximum generation reached by the search. We then looked at which generation the search found the last of its unique crash buckets. For the Media 1 searches, crash-finding searches seeded with well-formed files found all unique bugs within 4 generations, with a maximum number of generations between 5 and 7. Therefore, most of the bugs found by these searches are *shallow* — they are reachable in a small number of generations. The crash-finding Media 2 searches reached a maximum generation of 3, so we did not observe a trend here.

Figure 8 shows histograms of both crashing and non-crashing ("NoIssues") test cases by generation for Media 1 seeded with `wff-4`. We can see that most tests executed were of generations 4 to 6, yet all unique bugs can be found in generations 1 to 4. The number of test cases tested with no issues in later generations is high, but these new test cases do not discover distinct new bugs. This behavior was consistently observed in almost all our experiments, especially the "bell curve" shown in the histograms. This generational search did not go beyond generation 7 since it still has many candidate input tests to expand in smaller generations and since many tests in later generations have lower scores.

No clear correlation between coverage and crashes. We measured the absolute number of blocks covered after running each test, and we compared this with the locations of the first test case to exhibit each distinct stack hash for a crash. Figure 9 shows the result for a Media 1 search seeded with `wff-4`; the vertical bars mark where in the search crashes with new stack hashes were discovered. While this graph suggests that an increase in coverage correlates with finding new bugs, we did not observe this universally. Several other searches follow the trends shown by the graph for `wff-2`: they found all unique bugs early on, even if code coverage increased later. We found this surprising, because we expected there to be a consistent correlation between

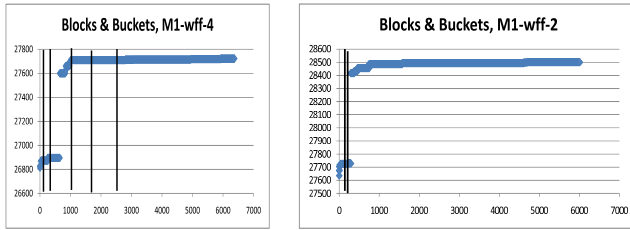

Figure 9: Coverage and initial discovery of stack hashes for Media 1 seeded with `wff-4` and `wff-2`. The leftmost bar represents multiple distinct crashes found early in the search; all other bars represent a single distinct crash first found at this position in the search.

new code explored and new bugs discovered. In both cases, the last unique bug is found partway through the search, even though crashing test cases continue to be generated.

Effect of block coverage heuristic. We compared the number of blocks added during the search between test runs that used our block coverage heuristic to pick the next child from the pool, and runs that did not. We observed only a weak trend in favor of the heuristic. For example, the Media 2 `wff-1` search added 10407 blocks starting from 48494 blocks covered, while the non-heuristic case started with 48486 blocks and added 10633, almost a dead heat. In contrast, the Media 1 `wff-1` search started with 27659 blocks and added 701, while the non-heuristic case started with 26962 blocks and added only 50. Out of 10 total search pairs, in 3 cases the heuristic added many more blocks, while in the others the numbers are close enough to be almost a tie. As noted above, however, this data is noisy due to non-determinism observed with code coverage.

5. OTHER RELATED WORK

Other extensions of fuzz testing have recently been developed. Most of those consist of using *grammars* for representing sets of possible inputs [28, 31]. Probabilistic weights can be assigned to production rules and used as heuristics for random test input generation. Those weights can also be defined or modified automatically using coverage data collected using lightweight dynamic program instrumentation [32]. These grammars can also include rules for corner cases to test for common pitfalls in input validation code (such as very long strings, zero values, etc.). The use of input grammars makes it possible to encode *application-specific knowledge* about the application under test, as well as *testing guidelines* to favor testing specific areas of the input space compared to others. In practice, they are often key to enable blackbox fuzzing to find interesting bugs, since the probability of finding those using pure random testing is usually very small. But writing grammars manually is tedious, expensive and scales poorly. In contrast, our whitebox fuzzing approach does not require an input grammar specification to be effective. However, the experiments of the previous section highlight the importance of the initial seed file for a given search. Those seed files could be generated using grammars used for blackbox fuzzing to increase their diversity. Also, note that blackbox fuzzing can generate and run new tests faster than whitebox fuzzing due to the cost of symbolic execution and constraint solving. As a result, it may be able to expose new paths that would not exercised with whitebox fuzzing because of the imprecision of symbolic execution.

As previously discussed, our approach builds upon recent work on systematic dynamic test generation, introduced in [14, 6] and extended in [13, 29, 7, 12, 27]. The main differences are that we use a generational search algorithm using heuristics to find bugs as fast as possible in an incomplete search, and that we test large applications instead of unit test small ones, the latter being enabled by a trace-based x86-binary symbolic execution instead of a source-based approach. Those differences may explain why we have found more bugs than previously reported with dynamic test generation.

Symbolic execution is also a key component of *static program analysis*, which has been applied to x86 binaries [2]. Static analysis is usually more efficient but less precise than dynamic analysis and testing, and their complementarity is well known [10, 13]. They can also be combined [13, 15]. *Static test generation* [19] consists of analyzing a program statically to attempt to compute input values to drive it along specific program paths *without ever executing the program*. In contrast, *dynamic* test generation extends static test generation with additional runtime information, and is therefore more general and powerful [14, 12]. Symbolic execution has also been proposed in the context of generating vulnerability signatures, either statically [5] or dynamically [9].

6. CONCLUSION

We introduced a new search algorithm, the *generational search*, for dynamic test generation that tolerates divergences and better leverages expensive symbolic execution tasks. Our system, SAGE, applied this search algorithm to find bugs in a variety of production machine-code x86 programs running on Windows. We then ran experiments to better understand the behavior of SAGE on two media parsing applications. We found that using a wide variety of well-formed input files is important for finding distinct bugs. We also observed that the number of generations explored is a better predictor than block coverage of whether a test case will find a unique new bug, and in particular that most unique bugs can be found within a small number of generations.

While these observations must be treated with caution, coming from a limited sample size, they suggest a new search strategy: instead of running for a set number of hours, systematically search a small number of generations starting from an initial seed file; once these test cases are exhausted, move on to a new seed file. The promise of this strategy is that it may cut off the "tail" of a SAGE search that only finds new examples of previously seen bugs. This would allow us to find more distinct bugs in the same amount of time. Future work could pursue this search method, possibly combining it with our block-coverage heuristic taken over different seed files to avoid re-exploring the same code multiple times. The key point to investigate is whether generation depth combined with code coverage is a better indicator of when to stop testing than code coverage alone.

Finally, we plan to enhance the precision of SAGE symbolic execution and the power of SAGE's constraint solving capability. This will enable SAGE to find bugs that are currently out of reach.

7. ACKNOWLEDGMENTS

We are indebted to Chris Marsh and Dennis Jeffries for important contributions to SAGE, and to Hunter Hudson for championing this project from the very beginning. SAGE builds on the work of the TruScan team, including Andrew Edwards and Jordan Tigani, and the Disolver team, including Youssf Hamadi and Lucas Bordeaux, for which we are grateful. We thank Tom Ball, Manuvir Das and Jim Larus for their support and feedback on this project. Various internal test teams provided valuable feedback during the development of SAGE, including some of the bugs described in Section 4.1, for which we thank them. We thank Derrick Coetzee and Ben Livshits for their comments on drafts of our paper, and Nikolaj Bjorner and Leonardo de Moura for discussions on constraint solving. We thank Chris Walker for helpful discussions regarding security.

8. REFERENCES

[1] D. Aitel. The advantages of block-based protocol analysis for security testing, 2002. `http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html`.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Int. Conf. on Compiler Construction*, 2004. `http://www.cs.wisc.edu/wpis/papers/cc04.ps`.

[3] S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments VEE*, 2006.

[4] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] D. Brumley, T. Chieh, R. Johnson, H. Lin, and D. Song. RICH : Automatically protecting against integer-based vulnerabilities. In *NDSS - Symposium on Network and Distributed System Security*, 2007.

[6] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[8] Microsoft Corporation. AppVerifier, 2007. `http://www.microsoft.com/technet/prodtechnol/windows/appcompatibility/appverifier.mspx`.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, , and P. Barham. Vigilante: End-to-end containment of internet worms. In *Symposium on Operating Systems Principles (SOSP)*, 2005.

[10] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of WODA'2003 (ICSE Workshop on Dynamic Analysis)*, Portland, May 2003.

[11] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, August 2000.

[12] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.

[13] P. Godefroid and N. Klarlund. Software Model Checking: Searching for Computations in the Abstract or the Concrete (Invited Paper). In *Proceedings of IFM'2005 (Fifth International Conference on Integrated Formal Methods)*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32, Eindhoven, November 2005. Springer-Verlag.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.

[15] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*, 2006.

[16] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 219–227, September 2000.

[17] Y. Hamadi. Disolver: the distributed constraint solver version 2.44, 2006. `http://research.microsoft.com/~youssefh/DisolverWeb/disolver.pdf`.

[18] M. Howard. Lessons learned from the animated cursor security bug, 2007. `http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx`.

[19] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[20] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[21] R. Majumdar and K. Sen. Hybrid Concolic testing. In *Proceedings of ICSE'2007 (29th International Conference on Software Engineering)*, Minneapolis, May 2007. ACM.

[22] D. Molnar and D. Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors, 2007. UC Berkeley EECS, 2007-23.

[23] Month of Browser Bugs, July 2006. Web page: `http://browserfun.blogspot.com/`.

[24] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Languages Design and Implementation (PLDI)*, 2007.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[26] J. Offutt and J. Hayes. A Semantic Model of Program Faults. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 195–200, San Diego, January 1996.

[27] Pex. Web page: `http://research.microsoft.com/Pex`.

[28] Protos. Web page:
`http://www.ee.oulu.fi/research/ouspg/protos/`.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.

[30] A. Sotirov. Windows animated cursor stack overflow vulnerability, 2007.
`http://www.determina.com/security.research/vulnerabilities/ani-header.html`.

[31] Spike. Web page:
`http://www.immunitysec.com/resources-freesoftware.shtml`.

[32] M. Vuagnoux. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany*, 2005. `autodafe.sourceforge.net`.

Media 1:	wff-1	wff-1-nh	wff-2	wff-2-nh	wff-3	wff-3-nh	wff-4	wff-4-nh
NULL	1 (46)	1 (32)	1(23)	1(12)	1(32)	1(26)	1(13)	1(1)
ReadAV	1 (40)	1 (16)	2(32)	2(13)	7(94)	4(74)	6(15)	5(45)
WriteAV	0	0	0	0	0	1(1)	1(3)	1(1)
SearchTime	10h7s	10h11s	10h4s	10h20s	10h7s	10h12s	10h34s	9h29m2s
AnalysisTime(s)	5625	4388	16565	11729	5082	6794	5545	7671
AnalysisTasks	564	545	519	982	505	752	674	878
BlocksAtStart	27659	26962	27635	26955	27626	27588	26812	26955
BlocksAdded	701	50	865	111	96	804	910	96
NumTests	6890	7252	6091	14400	6573	10669	8668	15280
TestsToLastCrash	6845	7242	5315	13616	6571	10563	6847	15279
TestsToLastUnique	168	5860	266	13516	5488	2850	2759	1132
MaxGen	6	6	6	8	6	7	7	8
GenToLastUnique	3 (50%)	5 (83%)	2 (33%)	7 (87.5%)	4 (66%)	3 (43%)	4 (57%)	3 (37.5%)
Mean Changes	1	1	1	1	1	1	1	1

Media 1:	wff-5	wff-5-nh	bogus-1	bogus-1-nh	bogus-2	bogus-3	bogus-4	bogus-5
NULL	1(25)	1(15)	0	0	0	0	0	0
ReadAV	3(44)	3(56)	3(3)	1(1)	0	0	0	0
WriteAV	0	0	0	0	0	0	0	0
SearchTime	10h8s	10h4s	10h8s	10h14s	10h29s	9h47m15s	5m23s	5m39s
AnalysisTime(s)	21614	22005	11640	13156	3885	4480	214	234
AnalysisTasks	515	394	1546	1852	502	495	35	35
BlocksAtStart	27913	27680	27010	26965	27021	27022	24691	24692
BlocksAdded	109	113	130	60	61	74	57	41
NumTests	4186	2994	12190	15594	13945	13180	35	35
TestsToLastCrash	4175	2942	1403	11474	NA	NA	NA	NA
TestsToLastUnique	1504	704	1403	11474	NA	NA	NA	NA
MaxGen	5	4	14	13	8	9	9	9
GenToLastUnique	3 (60%)	3 (75%)	10 (71%)	11 (84%)	NA	NA	NA	NA
Mean Changes	1	1	1	1	1	1	1	1

Media 2:	wff-1	wff-1-nh	wff-2	wff-3	wff-3-nh	wff-4	wff-4-nh	wff-5	wff-5-nh	bogus-1
NULL	0	0	0	0	0	0	0	0	0	0
ReadAV	4(9)	4(9)	0	4(15)	4(14)	4(6)	3(3)	5(14)	4(12)	0
WriteAV	0	0	0	0	0	0	0	1(1)	0	0
SearchTime	10h12s	10h5s	10h6s	10h17s	10h1s	10h3s	10h7s	10h3s	10h6s	10h13s
AnalysisTime(s)	3457	3564	1517	9182	8513	1510	2195	10522	14386	14454
AnalysisTasks	3	3	1	6	7	2	2	6	6	1352
BlocksAtStart	48494	48486	51217	41726	41746	48729	48778	41917	42041	20008
BlocksAdded	10407	10633	12329	9564	8643	10379	10022	8980	8746	14743
NumTests	1045	1014	777	1253	1343	1174	948	1360	980	4165
TestsToLastCrash	1042	989	NA	1143	1231	1148	576	1202	877	NA
TestsToLastUnique	461	402	NA	625	969	658	576	619	877	NA
MaxGen	2	2	1	3	2	2	2	3	2	14
GenToLastUnique	2 (100%)	2 (100%)	NA	2 (66%)	2 (100%)	2 (100%)	1 (50%)	2 (66%)	2 (100%)	NA
Mean Changes	3	3	4	4	3.5	5	5.5	4	4	2.9

Figure 10: Search statistics. For each search, we report the number of crashes of each type: the first number is the number of distinct buckets, while the number in parentheses is the total number of crashing test cases. We also report the total search time (SearchTime), the total time spent in symbolic execution (AnalysisTime), the number of symbolic execution tasks (AnalysisTasks), blocks covered by the initial file (BlocksAtStart), new blocks discovered during the search (BlocksAdded), the total number of tests (NumTests), the test at which the last crash was found (TestsToLastCrash), the test at which the last unique bucket was found (TestsToLastUnique), the maximum generation reached (MaxGen), the generation at which the last unique bucket was found (GenToLastUnique), and the mean number of file positions changed for each generated test case (Mean Changes). These numbers were generated with SAGE version 1.01.20416, built on 4/16/2007 at 4:39pm.