acmqueue

SAGE: Whitebox Fuzzing for Security Testing

SAGE has had a remarkable impact at Microsoft.

Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

Most *ACM Queue* readers might think of "program verification research" as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like 93-plus percent of PC users—that is, more than a billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

THE HIGH COST OF SECURITY BUGS

Every second Tuesday of every month, also known as "Patch Tuesday," Microsoft releases a list of security bulletins and associated security patches to be deployed on hundreds of millions of machines worldwide. Each security bulletin costs Microsoft and its users millions of dollars. If a monthly security update costs you $0.001 (one tenth of one cent) in just electricity or loss of productivity, then this number multiplied by a billion people is $1 million. Of course, if malware were spreading on your machine, possibly leaking some of your private data, then that might cost you much more than $0.001. This is why we strongly encourage you to apply those pesky security updates.

Many security vulnerabilities are a result of programming errors in code for parsing files and packets that are transmitted over the Internet. For example, Microsoft Windows includes parsers for hundreds of file formats.

If you are reading this article on a computer, then the picture shown in figure 1 is displayed on your screen after a jpg parser (typically part of your operating system) has read the image data, decoded it, created new data structures with the decoded data, and passed those to the graphics card in your computer. If the code implementing that jpg parser contains a bug such as a buffer overflow that can be triggered by a corrupted jpg image, then the execution of this jpg parser on your computer could potentially be hijacked to execute some other code, possibly malicious and hidden in the jpg data itself.

FIGURE 1

A Sample JPG Image

Source: NASA

This is just one example of a possible security vulnerability and attack scenario. The security bugs discussed throughout the rest of this article are mostly buffer overflows.

HUNTING FOR "MILLION-DOLLAR" BUGS

Today, hackers find security vulnerabilities in software products using two primary methods. The first is code inspection of binaries (with a good disassembler, binary code is like source code).

The second is *blackbox fuzzing*, a form of blackbox random testing, which randomly mutates well-formed program inputs and then tests the program with those modified inputs,[3] hoping to trigger a bug such as a buffer overflow. In some cases, *grammars* are used to generate the well-formed inputs. This also allows encoding application-specific knowledge and test-generation heuristics.

Blackbox fuzzing is a simple yet effective technique for finding security vulnerabilities in software. Thousands of security bugs have been found this way. At Microsoft, fuzzing is mandatory for every untrusted interface of every product, as prescribed in the Security Development Lifecycle,[7] which documents recommendations on how to develop secure software.

Although blackbox fuzzing can be remarkably effective, its limitations are well known. For example, the `then` branch of the conditional statement in

```
int foo(int x) { // x is an input
   int y = x + 3;
   if (y == 13) abort(); // error
   return 0;
}
```

has only 1 in 2^{32} chances of being exercised if the input variable `x` has a randomly chosen 32-bit value. This intuitively explains why blackbox fuzzing usually provides low code coverage and can miss security bugs.

INTRODUCING WHITEBOX FUZZING

A few years ago, we started developing an alternative to blackbox fuzzing, called *whitebox fuzzing*.[5] It builds upon recent advances in systematic dynamic test generation[4] and extends its scope from unit testing to whole-program security testing.

Starting with a well-formed input, whitebox fuzzing consists of *symbolically* executing the program under test *dynamically*, gathering constraints on inputs from conditional branches encountered along the execution. The collected constraints are then systematically negated and solved with a *constraint solver*, whose solutions are mapped to new inputs that exercise different program execution paths. This process is repeated using novel *search techniques* that attempt to sweep through all (in practice, many) feasible execution paths of the program while checking many properties simultaneously using a *runtime checker* (such as Purify, Valgrind, or AppVerifier).

For example, symbolic execution of the previous program fragment with an initial value 0 for the input variable `x` takes the `else` branch of the conditional statement and generates the path constraint `x+3 ≠ 13`. Once this constraint is negated and solved, it yields `x = 10`, providing a new input that causes the program to follow the `then` branch of the conditional statement. This allows us to exercise and test additional code for security bugs, even without specific knowledge of the input format. Furthermore, this approach automatically discovers and tests "corner cases" where programmers may fail to allocate memory or manipulate buffers properly, leading to security vulnerabilities.

In theory, systematic dynamic test generation can lead to full program path coverage, i.e., *program verification*. 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 can be imprecise due to complex program statements (pointer manipulations, floating-point operations, etc.), calls to external operating-system and library functions, and large numbers of constraints that cannot all be solved perfectly in a reasonable amount of time. Therefore, we are forced to explore practical tradeoffs.

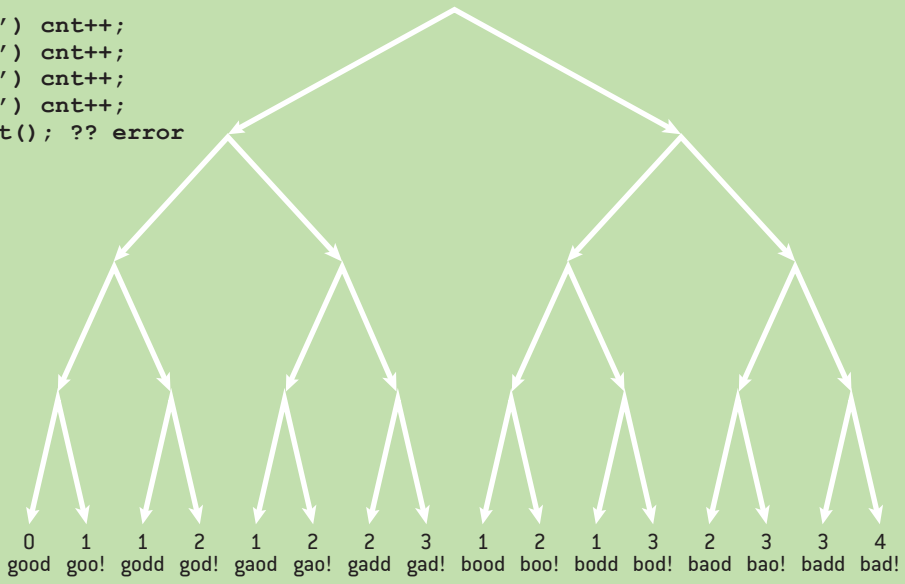
SAGE

Whitebox fuzzing was first implemented in SAGE (*Scalable Automated Guided Execution*).[5] Because SAGE targets large applications where a single execution may contain hundreds of millions of instructions, symbolic execution is its slowest component. Therefore, SAGE implements a novel directed-search algorithm, dubbed *generational search*, that maximizes the number of new input tests generated from each symbolic execution. Given a path constraint, *all* the constraints in that path are systematically negated one by one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver. This way, a single symbolic execution can generate thousands of new tests. (In contrast, a standard depth-first or breadth-first search would negate only the last or first constraint in each path constraint and generate at most one new test per symbolic execution.)

The program shown in figure 2 takes four bytes as input and contains an error when the value of

FIGURE 2 **Example of Program (Left) and Its Search Space (Right) with the Value of cnt at the End of Each Run**

```
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 >= 4) abort(); ?? error
}
```



the variable `cnt` is greater than or equal to four. Starting with some initial input `good`, SAGE runs this program while performing a symbolic execution dynamically. Since the program path taken during this first run is formed by all the `else` branches in the program, the path constraint for this initial run is the conjunction of constraints $i_0 \neq b$, $i_1 \neq a$, $i_2 \neq d$ and $i_3 \neq !$. Each of these constraints is negated one by one, placed in a conjunction with the prefix of the path constraint leading to it, and then solved with a constraint solver. In this case, all four constraints are solvable, leading to four new test inputs. Figure 2 also shows the set of all feasible program paths for the function `top`. The leftmost path represents the initial run of the program and is labeled 0 for Generation 0. Four Generation 1 inputs are obtained by systematically negating and solving each constraint in the Generation 0 path constraint. By repeating this process, all paths are eventually enumerated for this example. In practice, the search is typically incomplete.

SAGE was the first tool to perform dynamic symbolic execution at the x86 binary level. It is implemented on top of the trace replay infrastructure TruScan,[8] which consumes trace files generated by the iDNA framework[1] and virtually reexecutes the recorded runs. TruScan offers several features that substantially simplify symbolic execution, including 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. Thanks to offline tracing, constraint generation in SAGE is completely deterministic because it works with an execution trace that captures the outcome of all nondeterministic events encountered during the recorded run. Working at the x86 binary level allows SAGE to be used on any program regardless of its source language or build process. It also ensures that *"what you fuzz is what you ship,"* as compilers can perform source-code changes that may affect security.

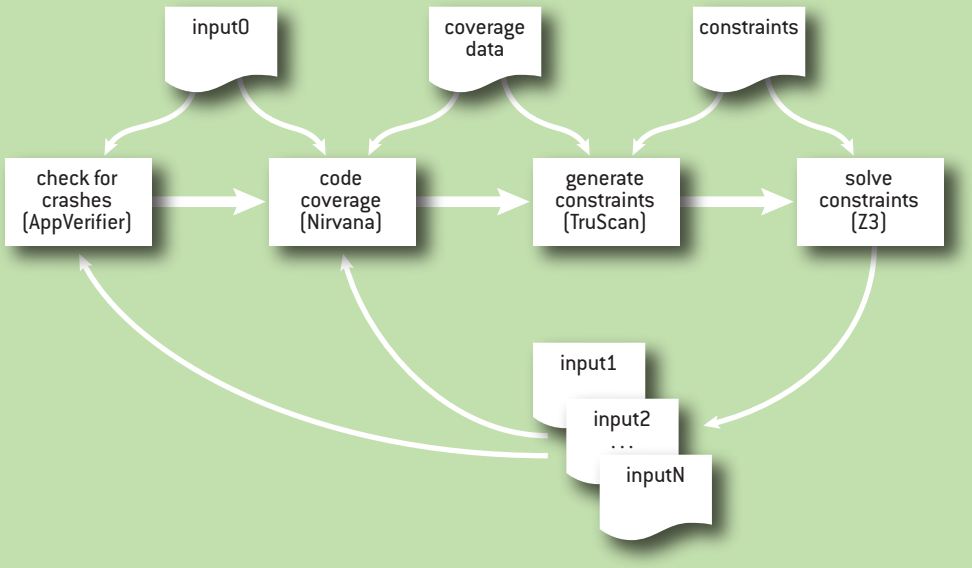
SAGE uses several optimizations that are crucial for dealing with huge execution traces. For example, a single symbolic execution of Excel with 45,000 input bytes executes nearly 1 billion x86 instructions. To scale to such execution traces, SAGE uses several techniques to improve the speed and memory usage of constraint generation: *symbolic-expression caching* ensures that structurally equivalent symbolic terms are mapped to the same physical object; *unrelated constraint elimination* reduces the size of constraint solver queries by removing the constraints that do not share symbolic variables with the negated constraint; *local constraint caching* skips a constraint if it has already been added to the path constraint; *flip count limit* establishes the maximum number of times a constraint generated from a particular program branch can be flipped; using a cheap syntactic check, *constraint subsumption* eliminates constraints logically implied by other constraints injected at the same program branch (most likely resulting from successive iterations of an input-dependent loop).

SAGE ARCHITECTURE

The high-level architecture of SAGE is depicted in figure 3. Given one (or more) initial input `Input0`, SAGE starts by running the program under test with AppVerifier to see if this initial input triggers a bug. If not, SAGE then collects the list of unique program instructions executed during this run. Next, SAGE symbolically executes the program with that input and generates a path constraint, characterizing the current program execution with a conjunction of input constraints.

Then, implementing a generational search, all the constraints in that path constraint are negated one by one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver (we currently use the Z3 SMT solver[2]). All satisfiable

FIGURE 3 **Architecture of SAGE**



constraints are mapped to `N` new inputs, which are tested and ranked according to incremental instruction coverage. For example, if executing the program with new `Input1` discovers 100 new instructions, then `Input1` gets a score of 100, and so on. The new input with the highest score is selected to go through the (expensive) symbolic execution task, and the cycle is repeated, possibly forever. Note that all the SAGE tasks can be executed in parallel on a multicore machine or even on a set of machines.

Building a system such as SAGE poses many other challenges: how to recover from imprecision in symbolic execution, how to check many properties together efficiently, how to leverage grammars (when available) for complex input formats, how to deal with path explosion, how to reason precisely about pointers, how to deal with floating-point instructions and input-dependent loops. Space constraints prevent us from discussing these challenges here, but the authors' Web pages provide access to other papers addressing these issues.

AN EXAMPLE

On April 3, 2007, Microsoft released an out-of-band critical security patch (MS07-017) 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.[9] This was only the third such out-of-band patch released by Microsoft since January 2006, indicating the seriousness of the bug. The Microsoft SDL Policy Weblog stated that extensive blackbox fuzzing of this code failed to uncover the bug and that existing static-analysis tools were not capable of finding the bug without excessive false positives.[6]

SAGE, in contrast, synthesized a new input file exhibiting the bug within hours of starting from a well-formed ANI file, despite having no knowledge of the ANI format. A seed file was picked

arbitrarily from a library of well-formed ANI files, and SAGE was run on a small test program that called `user32.dll` to parse ANI files. The initial run generated a path constraint with 341 branch constraints after executing 1,279,939 total x86 instructions over 10,072 symbolic input bytes. SAGE then created a crashing ANI file after 7 hours 36 minutes and 7,706 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.

IMPACT OF SAGE

Since 2007, SAGE has discovered many security-related bugs in many large Microsoft applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly *one-third* of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7. Because SAGE is typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing.

Finding all these bugs has saved millions of dollars to Microsoft, as well as to the world in time and energy, by avoiding expensive security patches to more than 1 billion PCs. The software running on *your* PC has been affected by SAGE.

Since 2008, SAGE has been running 24/7 on an average of 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs. This is more than 300 machine-years and the *largest computational usage ever for any SMT (Satisfiability Modulo Theories) solver*, with more than 1 billion constraints processed to date.

SAGE is so effective at finding bugs that, for the first time, we faced "bug triage" issues with dynamic test generation. We believe this effectiveness comes from being able to fuzz large applications (not just small units as previously done with dynamic test generation), which in turn allows us to find bugs resulting from problems across multiple components. SAGE is also easy to deploy, thanks to x86 binary analysis, and it is fully automatic. SAGE is now used daily in various groups at Microsoft.

CONCLUSION

SAGE has had a remarkable impact at Microsoft. It combines and extends program analysis, testing, verification, model checking, and automated theorem-proving techniques that have been developed over many years.

Which is best in practice—blackbox or whitebox fuzzing? Both offer different cost/precision tradeoffs. Blackbox is simple, lightweight, easy, and fast but can yield limited code coverage. Whitebox is smarter but more complex.

Which approach is more effective at finding bugs? It depends. If an application has never been fuzzed, any form of fuzzing is likely to find bugs, and simple blackbox fuzzing is a good start. Once the low-hanging bugs are gone, however, fuzzing for the next bugs has to become smarter. Then it's time to use whitebox fuzzing and/or user-provided guidance, for example, using an input grammar.

The bottom line? In practice, use both. We do at Microsoft.

ACKNOWLEDGMENTS

Many people across different groups at Microsoft have contributed to SAGE's success. Special thanks go to Ella Bounimova, Chris Marsh, Lei Fang, Stuart de Jong, and Hunter Hudson. SAGE builds on the work of the TruScan team—including Andrew Edwards and Jordan Tigani, as well as Evan Tice,

David Grant, and Vince Orgovan—and of the Z3 team, including Nikolaj Bjorner and Leonardo de Moura, as well as Youssef Hamadi and Lucas Bordeaux—for which we are grateful. SAGE would not exist without the extraordinary large-scale deployment and usage made possible by work done in the Windows security testing team, including Nick Bartmon, Eric Douglas, Dustin Duran, Elmar Langholz, and Dave Weston, and the Office security testing team, including Tom Gallagher, Eric Jarvi, and Octavian Timofte. We are also thankful to our MSEC colleagues Dan Margolis, Matt Miller, and Lars Opstad. SAGE also benefited from contributions of many talented research interns, namely Dennis Jeffries, Adam Kiezun, Bassem Elkarablieh, Marius Nita, Cindy Rubio-Gonzalez, Johannes Kinder, Daniel Luchaup, Nathan Rittenhouse, Mehdi Bouaziz, and Ankur Taly. We thank our managers for their support and feedback on this project. We also thank all other SAGE users across Microsoft.

REFERENCES

1. Bhansali, S., Chen, W., De Jong, S., Edwards, A., Drinic, M. 2006. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*.
2. de Moura, L., Bjorner, N. 2008. Z3: an efficient SMT solver. In *Proceedings of TACAS (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*: 337–340. Springer-Verlag.
3, Forrester, J. E., Miller, B. P. 2000. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th Usenix Windows System Symposium*, Seattle (August).
4. Godefroid, P., Klarlund, N., Sen, K. 2005. DART: Directed Automated Random Testing. In *Proceedings of PLDI (Programming Language Design and Implementation)*: 213–223.
5. Godefroid, P., Levin, M. Y., Molnar, D. 2008. Automated whitebox fuzz testing. In *Proceedings of NDSS (Network and Distributed Systems Security)*: 151– 166.
6. Howard, M. 2007. Lessons learned from the animated cursor security bug; http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-fromthe-animated-cursor-security-bug.aspx.
7. Howard, M., Lipner, S. 2006. *The Security Development Lifecycle*. Microsoft Press.
8. Narayanasamy, S.,Wang, Z., Tigani, J., Edwards, A., Calder, B. 2007. Automatically classifying benign and harmful data races using replay analysis. In *Programming Languages Design and Implementation (PLDI)*.
9 Sotirov, A. 2007. Windows animated cursor stack overflow vulnerability; http://www.determina.com/security.research/vulnerabilities/ani-header.html.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PATRICE GODEFROID is a principal researcher at Microsoft Research. He received a B.S. degree in electrical engineering (computer science elective) and a Ph.D. degree in computer science from the University of Liege, Belgium, in 1989 and 1994, respectively. From 1994 to 2006, he worked at Bell Laboratories. His research interests include program specification, analysis, testing, and verification. pg@microsoft.com.

MICHAEL Y. LEVIN is a principal development manager in the Windows Azure Engineering

Infrastructure team where he leads a team developing the Windows Azure Monitoring and Diagnostics Service. His additional interests include automated test generation, anomaly detection, data mining, and scalable debugging in distributed systems. He received his Ph.D. in computer science from the University of Pennsylvania. mlevin@microsoft.com.

DAVID MOLNAR is a researcher at Microsoft Research, where his interests focus on software security, electronic privacy, and cryptography. He earned a Ph.D. in 2009 from the University of California at Berkeley, working with David Wagner. dmolnar@microsoft.com.