Billions and Billions of Constraints: Whitebox Fuzz Testing in Production

Ella Bounimova   Patrice Godefroid   David Molnar

Abstract

We report experiences with constraint-based whitebox fuzz testing in production across hundreds of large Windows applications, multiple Microsoft product releases, and over 400 machine years of computations from 2007 to 2012. Whitebox fuzzing leverages symbolic execution on binary traces and constraint solving to construct new inputs to a program. These inputs execute previously uncovered paths or trigger security vulnerabilities. Whitebox fuzzing has found one-third of all file fuzzing bugs during the development of Windows 7, saving millions of dollars in potential security vulnerabilities. We present two new systems developed to manage our deployments: SAGAN , which collects data from every single fuzzing run for further analysis, and JobCenter, which controls deployment of our whitebox fuzzing infrastructure across commodity virtual machines. Since June 2010, SAGAN has recorded over 3.4 billion constraints solved, millions of symbolic executions, and tens of millions of test cases generated. Our work represents the largest scale deployment of whitebox fuzzing, including the largest computational usage ever for a Satisfiability Modulo Theories (SMT) solver, to date. We highlight specific improvements to whitebox fuzzing driven by our data collection and open problems that remain.

1 Introduction

Fuzz testing is the process of repeatedly feeding modified inputs to a program in order to uncover security bugs, such as buffer overflows. First introduced by Miller et al. [20], traditional *blackbox fuzzing* is a form of random testing where an initial *seed* input is randomly mutated to generate new *fuzzed* inputs. While simple, the technique has been shown to be startingly effective, as thousands of security vulnerabilities have been found this way. At Microsoft, fuzz testing is now required by the Security Development Lifecycle for code that may handle untrusted input.

Unfortunately, random testing often misses bugs that depend on specific trigger values or special structure to an input. More recently, advances in symbolic execution, constraint generation and solving have enabled *whitebox* fuzzing [18], which builds upon recent advances in systematic dynamic test generation [16, 5], 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. Each of those constraints are then 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 systematic *state-space search techniques*, inspired by *model checking*, that attempt to sweep through as many as possible feasible execution paths of the program while checking simultaneously many properties using a *runtime checker* (such as Purify, Valgrind or AppVerifier).

In this paper, we report on the usage of whitebox fuzzing on a very large scale for the first time. Earlier applications of dynamic test generation focused on unit testing of small programs [16, 5, 24], typically consisting of a few thousand lines of code, for which these techniques were able to achieve high code coverage and find new bugs, for instance, in Unix utility programs [4] or device drivers [7]. While promising, this prior work did not report of any daily use of these techniques and tools.

In contrast, we present here our experience running whitebox fuzzing on a much *larger scale* and in *production* mode. Our work follows from one simple but key observation: the *"killer app"* for dynamic test generation is *whitebox fuzzing of file parsers*. Many security vulnerabilities are due to programming errors in code for parsing files and packets that are transmitted over the internet. For instance, the Microsoft Windows operating system includes parsers for hundreds of file formats. Any security vulnerability in any of those parsers may require the deployment of a costly visible security patch to more than a billion PCs worldwide, i.e., millions of dollars.

Today, our whitebox fuzzer SAGE is now running every day on an average of 200 machines, and has been running for over 400 machine years since 2008. In the process, it has found many previously-unknown security vulnerabilities in hundreds of 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.

Achieving this milestone required facing key challenges in *scalability* across several dimensions:

Symbolic execution: how to efficiently perform symbolic execution on x86 execution traces with hundreds of millions of instructions and tens of thousands of symbolic variables for applications with millions of lines of code (like Microsoft Excel).

Constraint generation and solving: how to generate, solve and manage billions of constraints.

Long-running state-space searches: how to perform systematic state-space explorations (akin model checking) effectively for weeks or months at a time.

Diversity: how to easily configure, check and monitor whitebox fuzzing so that it is applicable to hundreds of diverse applications.

Fault tolerance and always-on usage: how to manage hundreds of machines running whitebox fuzzing 24/7 with as little down-time as possible.

While the first challenge was previously discussed in [18], we describe for the first time how to address the other four challenges.

In addition, we faced challenges from running in *production*, instead of in a research environment. Specifically, large product groups at Microsoft, such as the Windows and Office divisions, have large fuzzing labs of machines allocated to fuzz testing. Over the last few years, our whitebox fuzzer was *gradually* deployed on a larger and larger scale in those labs which were not under our control. Indeed, our internal customers were not mandated to use this new technology. Instead, we had to progressively gain their trust and business, by consistently finding new security-critical bugs in many applications, and missed by other competing fuzzers, while growing our initial research prototype to a full-fledged large scale operation over several years. For this, we had to gain a better understanding of how our whitebox fuzzer is being used and configured, as well as monitor its progress, while addressing the scalability challenges listed above and continually improving our tool and infrastructure.

We have developed and deployed two new systems to help meet these challenges: SAGAN and *JobCenter*. SAGAN is a monitoring system which records information from every SAGE run and displays it via a Web site, making it easy to drill down into the progress of a run. Since June 2010, SAGAN has recorded over 3.4 billion constraints solved, millions of symbolic executions, and tens of millions of test cases generated for hundreds of applications.

JobCenter is a control system which can auto-assign SAGE jobs to machines as they become available and monitor progress of each run. JobCenter also helps us manage the complexity of different configurations required for different programs under test.

Our infrastructure enables *data-driven improvement*, where feedback from previous runs helps us focus limited resources on further research and improve future runs. In Section 4, we demonstrate this with analyses enabled by SAGAN data that have led directly to changes in our whitebox fuzzing practice.

This paper is organized as follows. In Section 2, we review whitebox fuzzing and SAGE . In Section 3, we present our two new systems, SAGAN and JobCenter, and their main features. Then we present in Section 4 several original analyses on the performance of whitebox fuzz testing for runs of multiple weeks on many different programs. Each of those analyses were designed to lead to concrete actionable items that led to further improvements in our tool and infrastructure. We discuss other related work in Section 5, before concluding in Section 6.

2 Background: Whitebox Fuzzing

2.1 Blackbox Fuzzing

Blackbox fuzzing is a form of *blackbox random* testing which randomly mutates well-formed program inputs, and then tests the program with those modified inputs [13] with the hope of triggering a bug like a buffer overflow. In some cases, *grammars* are used to generate the well-formed inputs, which 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" [8] which documents recommendations on how to develop secure software.

Although blackbox fuzzing can be remarkably effective, its limitations are well-known. For instance, 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.

2.2 Whitebox Fuzzing

Whitebox fuzzing [18] is an alternative approach, which builds upon recent advances in systematic dynamic test generation [16, 5], 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 search techniques that attempt to sweep through all (in practice, many) feasible execution paths of the program while checking simultaneously many properties using a runtime checker.

For example, symbolic execution of the above 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 $x + 3 \neq 13$. Once this constraint is negated and solved, it yields $x = 10$, which gives us 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 properly allocate memory or manipulate buffers, 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 which cannot all be solved perfectly in a reasonable amount of time. Therefore, we are forced to explore *practical tradeoffs*.

# instructions executed	1,455,506,956
# instr. executed after 1st read from file	928,718,575
# constraints generated (full path constraint)	25,958
# constraints dropped due to cache hits	244,170
# constraints dropped due to limit exceeded	193,953
# constraints satisfiable (= # new tests)	2,980
# constraints unsatisfiable	22,978
# constraint solver timeouts (>5 secs)	0
symbolic execution time (secs)	2,745
constraint solving time (secs)	953

Figure 1: Statistics for a single symbolic execution of a large Office application with a 47 kilobyte input file.

2.3 SAGE

The basis of our work is the whitebox fuzzer SAGE [18]. Because we target large applications where a single execution may contain hundreds of millions of instructions, symbolic execution is the slowest component. Therefore, we use a *generational search* strategy to maximize 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.)

To give the reader an idea of the sizes of path constraints for large applications, Figure 1 shows some statistics about a single sample symbolic execution of a large Office application while parsing an input file of about 47 kilobytes. For file parser fuzzing, each byte read off the untrusted input file corresponds to a symbolic variable. Our whitebox fuzzer uses several optimizations that are crucial for dealing with such huge execution traces. These optimizations are discussed later in Section 4.

Our whitebox fuzzer performs dynamic symbolic execution at the x86 binary level. It is implemented on top of the trace replay infrastructure TruScan [23] which consumes trace files generated by the iDNA framework [1] and virtually re-executes 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

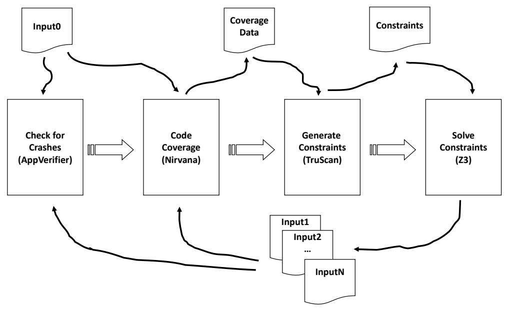

Figure 2: Architecture of SAGE .

stack frame allocations, and tracking the flow of data through the program structures. Thanks to off-line 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 our tool 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 which may impact security.

2.4 SAGE Architecture

The high-level architecture of SAGE is depicted in Figure 2. Given an 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 [9]). All satisfiable constraints are then mapped to N new inputs. These N new inputs are then tested and ranked according to incremental instruction coverage. For instance, if executing the program with new Input1 discovers 100 new instructions, Input1 gets a score of 100, and so on. Next, 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 multi-core machine or even on a set of machines; we discuss this in the next section.

3 Infrastructure

3.1 SAGAN

On a single machine, a multi-week whitebox fuzz testing run can consume hundreds of gigabytes of disk, perform thousands of symbolic executions and create many test cases. Each task in the SAGE pipeline offers opportunities for something to go wrong, potentially causing the fuzz run to stop in its tracks. Moreover, each task also offers room for improvements by observing statistics, starting with the basic time taken in each task. When we first started running whitebox fuzzing at scale, we had no way to capture this information, short of logging in remotely to computers that had participated in fuzz runs. Even then, to avoid running out of disk space, we typically configured SAGE to delete intermediate results. As a result, we had limited ability to detect failures and learn from our test runs.

3.1.1 Principles

We designed a logging service called SAGAN to address those issues. We developed the following key principles when designing our logging.

- First, *every run of* SAGE *creates a unique log.* Even if the run fails to start properly, we assign a globally unique identifier to the run. This allows us to unambiguously identify specific SAGE runs, aiding our debugging and statistics gathering.
- Second, *every log contains enough information to reproduce the run*, including all configuration files and command line options. This principle allows us to quickly reproduce failing runs and search for the cause of a problem. In the case of failed tasks, we also send back stdout and stderr files, which helps us diagnose previously unseen errors.
- Third, *every log has a unique URL that exposes the log information in a web browser.* We have found this simplifies collaborative troubleshooting and brainstorming over our next directions, because we can simply e-mail relevant links back and forth. We have also created a summary web front end that shows recent runs and summary statistics, such as number of crashing test cases found or number of runs that have a certain percentage of tasks failing. Using this front end, we can quickly check ongoing lab runs and highlight those with failing tasks or those which are finding many crashing test cases. In cases where users of SAGE need to change their

configuration, we can then send them the URL for the log and highlight the problem.

- Fourth, the logging infrastructure should be *low impact for the client.* This starts with an infrastructure that takes a small amount of overhead for each client. Beyond this, it means that even if the logging server fails, we ensure that the SAGE run can stll continue.
- Finally, *the central logs contain enough information for all analyses.* Prior to SAGAN , we would perform analyses in support of our research by running SAGE on a target of interest, then investigating files on disk to gather statistics and test hypotheses. While this could be scripted, it was necessarily time consuming because it could be done only on a machine by machine basis. In addition because this operation required running SAGE in a way that kept all intermediate data, it necessarily was not the same as the real test runs. With SAGAN , we can perform analyses, including all the analyses reported in this paper, without ever touching individual machines.

3.1.2 Architecture

We built the SAGAN logging service to support hundreds of simultaneously active machines. A central server runs Microsoft SQL Server and Internet Information Server on Windows Server 2008 R2. Each of the client machines makes a direct connection to the SQL Server to insert log updates. Updates happen at the beginning of every SAGE run, after every new crashing test case, after every failed task, and then at random intervals ranging between 30 to 45 minutes during the run. We randomize the intervals for updates to avoid synchronization of periodic messages, as recommended by Floyd and Jacobson [12].

While the overall disk usage of a SAGE run can total hundreds of gigabytes, we applied our principles to reduce the amount of information that must be shipped to our central server. First, we limit the initial information sent to the configuration options and command line, which are a few kilobytes each. Second, each heartbeat contains counters representing the number of files created, number of crashes found, coverage, and a small log file from our whitebox fuzzer, which is also typically under 10 kilobytes.This means that for a run with no failed tasks, no constraint solver timeouts, and no crashing test cases found, our space and bandwidth requirements are modest. For constraint solver timeouts, while we do ship the entire constraint to SAGAN for later analysis, we limit the number of such constraints shipped on each run. Finally, while in the case of failed symbolic execution tasks we may need to ship instruction traces in the hundreds of megabytes, we probabilistically decide whether or not to ship individual traces and ship at most 5 such traces per run.

3.1.3 Data Presentation

We present the data in two main ways. First, we designed a web front end to expose information from the server. Every run has its own unique URL that shows configuration information, health, and number of crashes found at a glance. By clicking on a link, we can drill down into statistics from any of the symbolic execution tasks that have completed during that fuzzing run. For example, we can see how many constraints were generated and how often optimizations in constraint generation were invoked.

Second, we run SQL queries against the tables holding data from SAGE runs. This gives us the flexibility to answer questions on the fly by looking at the data. For example, we can create a list of every SAGE run that has at least one symbolic execution task where more than 10 queries to Z3 timed out. We use this capability to work with our partners and understand if there are specific features of programs that might cause long-running constraint solver queries or other strange behavior. All the analyses in Section 4 were performed using our infrastructure.

3.2 JobCenter

The SAGAN system gives us insight, but it is only one piece of the puzzle. We need an active control system for the machines running SAGE . In particular, the programs on which we test SAGE are not static. While SAGE is running, developers are continually updating the code, fixing bugs and adding new features. Periodically we must upgrade the programs SAGE tests to the most recent version, to ensure that the bugs SAGE finds are most likely to reproduce on developers' machines and have not been fixed already.

3.2.1 Configuration Management

We run SAGE against multiple *configurations.* A configuration consists of a set of initial test cases, or *seed files*, a target program for execution with its arguments, and a set of parameters that define timeout values for each of the different SAGE tasks. One program of interest may have many configurations. For example, a single large program may have parsers embedded for many different file formats. Together with our partners who run SAGE , we have defined hundreds of distinct SAGE configurations in use today.

Manually starting SAGE with correct configurations on hundreds of machines would be a nightmare. We designed a control service called *JobCenter* that automates this work, in conjunction with additional infrastructure created by different partners of SAGE . Our partners typically have infrastructure that can automatically upgrade a machine to the latest build of the software under test and reset the disk to a clean state. JobCenter then deploys the right version of SAGE and the right configuration to start the fuzz testing run. While we still need manual effort to determine if a configuration has the right parameters and options, JobCenter allows us to do this work once and re-use it across multiple machines automatically.

3.2.2 Control and Recovery

We have made changes to SAGE that integrate it with JobCenter for runtime control. A JobCenter web service allows changing configuration values on the fly, which we use for fine-tuning and experimentation. For example, we can change how long to wait while tracing the execution of the test program. We can also pause a SAGE job, allowing us to upgrade the version of SAGE used or perform in depth analysis. We have implemented facilities for controlling multiple jobs at once, as well.

JobCenter can detect when a SAGE run has terminated, then trigger a request to JobCenter for a new configuration. This means that even if a particular configuration is faulty and leads to a paused run, we can try to recover and continue to use the machine; we developed this after looking at the utilization data in Section 4. We currently have at least 90 concurrent virtual machines reporting to JobCenter at any given time.

Finally, our test machines run Windows and typically need periodic reboots for security patches. Power outages, servicing, or other events may also cause unexpected reboots. We have modified SAGE to persist run information to disk in a way that ensures we can pick up after such an event when the machine comes back up. The JobCenter remembers which configuration is associated with a machine and on machine boot can re-start an interrupted whitebox fuzzing run. Prior to this, we had difficulty achieving runs of more than a month in length. Figure 3 shows the overall architecture, with VMs running SAGE talking to JobCenter and SAGAN .

3.3 Task Partitioning

As we discussed in Section 2, a SAGE fuzzing run consists of four different types of tasks. In principle, these tasks could be run on separate machines. For example, we could perform all tracing on one machine, then send the resulting instruction traces to a second machine for symbolic execution, forward the resulting constraints to another machine. Finally, we could run the newly created tests and measure coverage on a yet another machine.

In our runs, we typically keep all tasks on the same machine. We do this for two reasons. First, because the instruction-level traces we create are often hundreds of megabytes in size, which means we would incur delays in moving them between machines. We do not have the ability to control the network infrastructure used by our partners, so we cannot assume fast links between pairs of machines. Second, as we will see in Section 4, in our experience most constraints are solved within a tenth of a second, meaning that it is cheaper to solve them locally on the machine than to serialize, send over the network, deserialize, and solve.

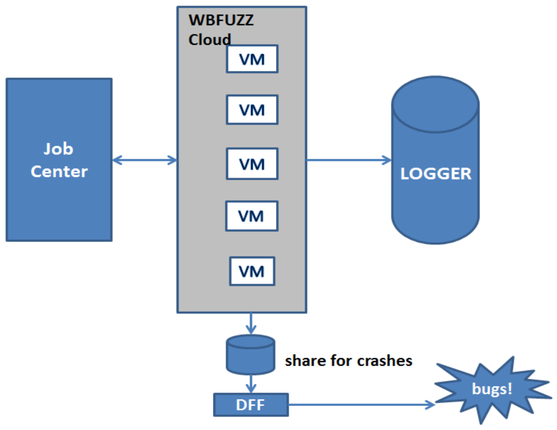

Figure 3: Architecture for JobCenter and SAGAN . Machines running SAGE communicate to SAGAN for logging and JobCenter for control. Crashing test cases are placed on a network share, where they are picked up by the Distributed File Fuzzer test farm. This test farm applies automatic triage to identify likely bugs, which humans then review and file in a bug database.

In Office, however, our partners have developed an infrastructure for leveraging "nights and weekends" of machines deployed around the company to test files for crashes [14]. To leverage this infrastructure, which can test hundreds of thousands of files in hours, we have a mode where SAGE copies crashing test cases on a network share for later processing. This infrastructure also performs an additional layer of triage, identifying bugs that may have already been filed, and prioritizing likely new bugs in front of users.

4 Data-Driven Whitebox Fuzzing

We now describe several specific analyses enabled by SAGAN data collected from runs of SAGE at scale. For each analysis, we describe improvements made to our fuzzing practice, or key insights that suggest future directions for whitebox fuzzing. The data we present spans several major *sessions* which each consist of hundreds of individual executions of the SAGE tool, each SAGE run itself fuzzing a different application during typically two to four weeks on a dedicated multi-core machine.

We note that the data analyzed here has been collected not from controlled experiments, but from production test runs. We did not choose which applications to test in each session, the length of the run, or the specific configuration settings. This means that sometimes, the data for different sessions might be more difficult to compare. Because the data that are being collected are so diverse, however, we were able to gain valuable feedback to track issues and evaluate improvements.

4.1 Utilization and Failure Detection

The first key analysis is monitoring utilization and detecting failures in SAGE deployments. First, we can determine failing runs by monitoring heartbeats. Before SAGAN , we had no way to know how the runs progressed, meaning that runs could die, perhaps due to a wrong test setup or running out of disk space, and it would not be apparent until we examined the results, which could be weeks later.

Using data from SAGAN , we improved our lab-machine utilization over time. We show here data for three successive sessions. In the first session with SAGAN , we were able to detect that many of the machines died unexpectedly, as shown on the leftmost chart of Figure 4. We then used the configuration files sent back by SAGAN to diagnose potential failures, then develop a fix. As a result, we saw improved utilization in the second and third major sessions, as shown on the middle and right charts of Figure 4. In another instance (data not shown), after several hundreds of runs had started (and were expected to run for at least three weeks), we were able to detect within hours that all of the symbolic execution tasks were failing due to a temporary networking problem that happened during the setup of the session. We then corrected the problem and re-started the session.

Second, besides obvious symptoms of a failing run, we check data on SAGAN that indicate how the run is proceeding. In particular we check whether the run generated any new tests, whether the symbolic execution task detected symbolic inputs, and how many bugs have been found. This in turn helps detect configuration errors. For example, if the timeout for tracing the application is set too low, then the trace will end before the application even reads from an input file. We then see this show up in SAGAN as a large percentage of symbolic execution tasks that fail to detect any symbolic inputs. For runs with problems, the JobCenter infrastructure allows adjusting configuration parameters of the run. In this example, we can increase the maximum time allowed for creating the execution trace to solve the problem.

SAGAN data also showed us that in the first and second sessions, 7% and 3% of all SAGE executions died due to the machines exhausting disk space. We modified SAGE to remove non-critical files automatically during low disk space conditions. All subsequent runs have had 0% failures due to low disk space.

We have also used SAGAN to detect which SAGE tasks have failed most, and why. For example, in one session, 62% of the about 300 SAGE runs had failures in symbolic execution which did not appear in the previous session. When we analyzed SAGAN data about these failures, we found a common problem and traced it to the changes in the compiler generating the binaries under test. Working with our partners, we fixed the problem, and in the next session, we had only 11% of all SAGE runs failing due to symbolic execution failures, with none of these remaining failures due to this compiler change.

4.2 Unique Bugs Found by Day

We also investigated when unique bugs are detected over the course of sessions. Our whitebox fuzzer SAGE uses AppVerifier configured to check for heap errors. Each memory corruption found (such as buffer overflow) is mapped to a crash. Because SAGE 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.

We collected earliest detection timestamps for each bucket found during a session. Figure 5 presents this chart for a session over two hundred programs over three weeks. We can see the first few days were the most productive, due to high number of new executions traced covered by symbolic execution[1]. The chart also shows two more ”peaks” of new crash buckets on days 13 and 21. This shows that new crashes were found throughout the session.

4.3 Incompleteness and Divergences

Another key analysis we performed was tracking incompleteness in symbolic execution. The x86 instruction set has over a thousand different instructions. New extensions are added frequently for supporting SIMD operations, such as Intel’s SSE instruction set. Unfortunately, SAGE does not understand how to symbolically execute every such instruction. This is important because failures

[1]Unfortunately, we cannot reveal absolute numbers of crashes found.

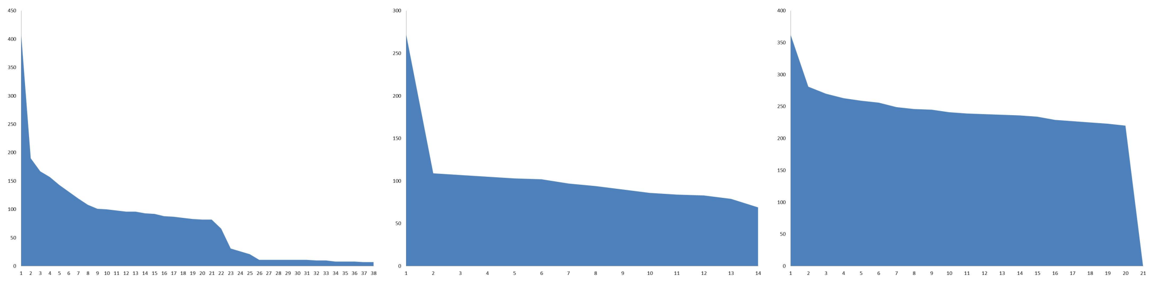

Figure 4: Utilization graph for three sequential sessions with SAGAN . The x-axis shows time in days, and the y-axis shows the number of active SAGE runs. By the third session, our utilization has improved due to underlying errors identified thanks to SAGAN data and then fixed.

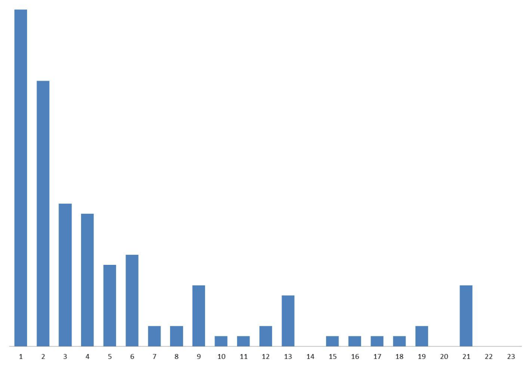

Figure 5: New unique crash buckets per day over 23 days of running SAGE on about 200 programs. The data suggests that running longer would yield more unique crashes, although the return becomes typically lower.

to properly symbolically execute can lead to an incomplete or wrong path constraint generation, with the effect of missing bugs or *divergences*: an input that is expected to drive the program along a new specific path actually follows a different path.

We added instrumentation to SAGE to detect whenever we found an instruction or an instruction sequence not properly handled by our symbolic execution engine. We then configured SAGE to send back counts of how many of such cases and of which type were encountered during every symbolic execution. Moreover, we prioritized the incompleteness cases into "high" and "low" severity categories. The high severity case indicates instructions not handled at all by our symbolic execution. Low severity means that the handling has known shortcomings but still creates some kind of approximate constraint.

After a first session with this new instrumentation, we analyzed the new resulting SAGAN data to determine which instructions and sequences of instructions had the highest counts. For example, we found that over 90% of the high severity instructions were shift instructions. We implemented symbolic instruction handlers for these instructions. As a result, the data from the next session showed that in the high severity category, the vast majority of the instructions had now been handled. We repeated this process across subsequent session, in order to address remaining holes in symbolic execution and prioritize the writing of new symbolic instruction handlers.

4.4 Constraint Generating and Solving

Another large amount of data we have collected during our 400 machine-years of running SAGE relates to symbolic execution and constraint solving. We now present such data for a sample set of about $300,000$ symbolic executions performed on about 300 different applications running on Windows, and their corresponding constraints. To be exact, this set consists of $304,190$ symbolic execution tasks. The sum of all constraints generated and solved during those symbolic executions is $129,648,907$ constraints, thus an average of 426 constraints generated for each symbolic execution (after all the constraint pruning done by the techniques and heuristics described later in this section).

4.4.1 Constraint Solving Time

Figure 6 presents the average solver time per constraint for each symbolic execution. From this chart, one can see that about 90% of all constraints are solved by Z3 in 0.1 seconds or less, and that about 99% of all constraints are solved in 1 second or less. Thus, most solver queries are fast.

4.4.2 Symbolic Execution and Solver Time

Figure 7 compares the total time in seconds spent symbolically executing programs versus solving constraints for each of the 300,000 symbolic execution tasks considered. Even though most constraints are solved in less than

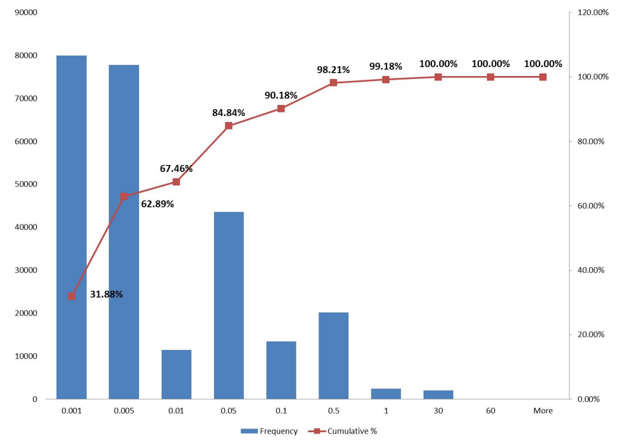

Figure 6: Average time for solving constraint queries for each symbolic execution task. The blue bars show the number of tasks with an average solving time in the labeled bin. The red line is a cumulative distribution function over all tasks. 90.18% of all tasks have an average solving time of 0.1 seconds or less.

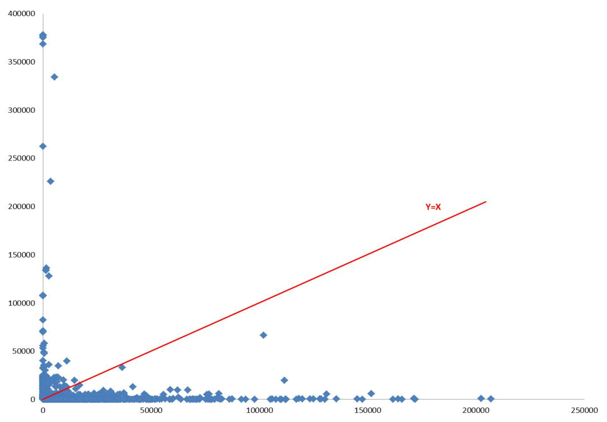

Figure 7: Each dot is a symbolic execution task. On the y-axis, the time to create constraints (secs). On the x-axis, the time to solve constraints (secs). Notice extreme outliers in both directions.

1 second, we can see that solving time often dominates symbolic execution time, because those runs solve many, many constraints. Moreover, we can also see extreme cases, or *outliers*, where symbolic execution time dominates strongly (upper left) or where solving time dominates exclusively (lower right).

A more detailed analysis of the data depicted in Figure 7 reveals that almost 95% of all the symbolic execution tasks fit in the lower left corner of the graph, in a box bounded by 200 seconds by 200 seconds. Zooming in that part of graph reveals the picture shown in Figure 8.

Following this observation, we implemented new features in SAGE to (1) limit the time in symbolic execution and (2) limit the number of constraints being generated. These new features can cut off the outliers observed in Figure 7, which consume a lot of CPU time but produce few constraints and hence few tests (upper left area) or spend too much time solving constraints (lower right area). By zooming in on tasks in the lower right area, we observed (data not shown here) that most of those constraints are actually unsatisfiable, and therefore do not contribute to new tests. Indeed, intuitively, the more symbolic execution generates constraints, the longer the path constraint (by definition), the more constrained the "tail" of the path constraint is (since every negated constraint is put in a conjunction with all the constraints before it in the path constraint), and the more unsatisfiable the constraints in the tail usually are. Therefore, most constraints in long tails are usually unsatisfiable. Of course, dropping unsatisfiable constraints is harmless for the purpose of test generation and finding bugs.

After enforcing such new limits on symbolic execution time and the number of constraints generated by each symbolic execution, we saw in the next session increases in the number of symbolic execution tasks per SAGE run (5.3 times) as expected, but also an increase in the average number of queries per symbolic execution task (2.3 times) and an increase in the total number of queries per SAGE run (12 times).

4.4.3 Unsatisfiable Constraints

Figure 9 plots the number of satisfiable constraints (x axis) versus the number of unsatisfiable constraints and timeouts (y axis) for all symbolic execution tasks in the previous data set that have a number of SAT solver queries less than 1,000 and a total number of queries less than 5,000. This set contains 298,354 tasks, which represents 98.08% of all tasks in the previous data set (the remaining outliers are hard to visualize in this form and were therefore omitted). This figure illustrates that most constraints generated by most symbolic executions are unsatisfiable – most dots are above the red line where $y = x$.

Why are most constraints generated by SAGE solved in a fraction of a second? An important optimization we use is *related constraint optimization* [18] which removes the constraints in the path constraint that do not share symbolic variables with the negated constraint (a simple syntactic form of "cone-of-influence" reduction); this optimization often eliminates more than 90% of the constraints in the path constraint. Another critical (and standard) optimization is *symbolic-expression caching* which

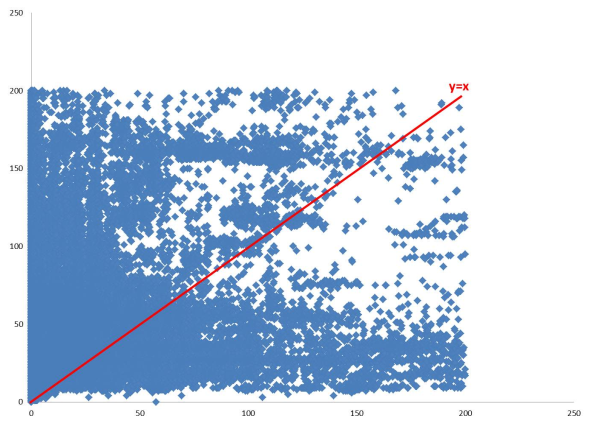

Figure 8: Symbolic execution tasks limited to a maximum of 200 seconds for constraint generation and 200 seconds for constraint solving. This set accounts for over 94 percent of all tasks shown in Figure 7.

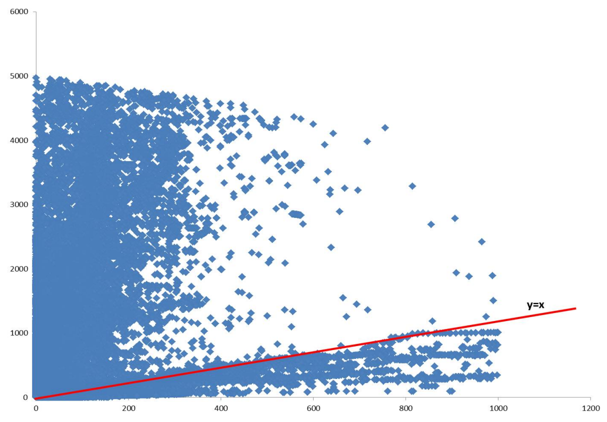

Figure 9: Each dot is a symbolic execution task. On the x-axis, the number of satisfiable constraints, i.e., the number of new test cases generated. On the y-axis, the number of unsatisfiable or timedout constraints. Most tasks have a high proportion of unsatisfiable constraints.

ensures that structurally equivalent symbolic terms are mapped to the same physical object, and avoids an exponential blow-up when the same sub-terms appear in different sub-expressions. We also use *local constraint caching* which skips a constraint if it has already been added to the path constraint (since its negation cannot possibly be satisfiable with the first constraint in the path constraint, i.e., $p \wedge \neg p$ is always unsatisfiable no matter what p is). These three optimizations are *sound*, that is, they cannot themselves cause to miss bugs.

We also use other optimizations which are necessary in practice to avoid the generation of too long path constraints but arbitrarily prune the search space and are therefore *unsound*, i.e., can force the search to miss bugs. Specifically, a *flip count limit* establishes the maximum number of times a constraint generated from a particular program branch can be flipped. Moreover, using a cheap syntactic check, *constraint subsumption* eliminates constraints logically implied by other constraints injected at the same program branch (mostly likely due to successive iterations of an input-dependent loop).

Together, these optimizations reduce the size and complexity of the queries sent to the constraint solver. Also, most of these queries are by construction essentially conjunctive (i.e., large conjunctions of constraints), and therefore known to be rather easily solvable most of the time. (In contrast, large disjunctions are usually harder to solve.) Our results show that, for whitebox fuzzing, *the art of constraint generation* is as important as *the art of constraint solving*.

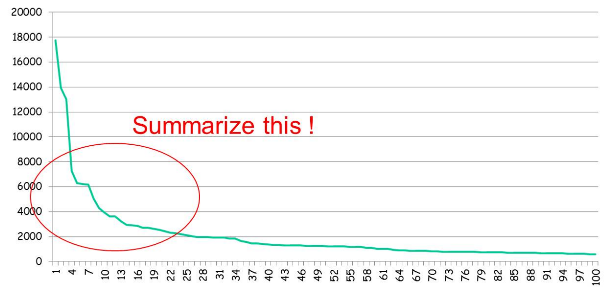

Figure 10: On the x-axis, we place different branches observed during whitebox fuzzing. On the y-axis we have the frequency of occurrence. The graph shows only the 100 first most common branches out of $3,360$ total.

4.5 Commonality Between Programs

When running a whitebox fuzzer at our scale on hundreds of applications, we are bound to have common subcomponents being re-fuzzed over and over again, especially in the Windows operating system where a lot of the core system functions are packaged in DLLs (dynamically loaded libraries) used by many of those applications.

Figure 10 shows statistics about all the program branches flipped during a whitebox fuzzing session for about 200 applications running on Windows. Each program branch is identified by a DLL name and an offset in that DLL, identifying a conditional statement (typically a jump) at that location, and where a symbolic input-dependent constraint was generated. The data represent 290,430 program branches flipped. There are 3,360 distinct branches, with a maximum frequency of 17,761 (extreme left) and minimum frequency 592 in the extreme right, which is not shown here – the tail is shown only up to distinct branch ranked as 100, and there is a very long and flat tail up to distinct branch 3,360 after this.

As one can clearly see from the figure, a small percentage of unique program branches (to the left) are flipped over and over again, and represent the part of the search space where most constraints were generated. Remember the data shown here was obtained *after* the pruning described in the previous subsection preventing the same branch to be flipped over a specific limit; without this pruning, the data would be even more tilted towards these few instructions. This behavior is typical, even in a single application. For instance, when whitebox fuzzing a structured non-binary parser, most of the input constraints generated are in the lexer, of course.

Re-fuzzing over and over again the same sub-components and DLLs is wasteful. In principle, this can be avoided with *compositional testing* [15], which creates *test summaries* from symbolic execution. These summaries are not only re-usable during a fuzzing session, but also apply across applications that share common components (such as DLLs), and *over time* from one fuzzing session to the next [17]. Compositional testing can result in a search algorithm that is *exponentially* faster than a non-compositional one. Every test run in a centralized infrastructure can create new test summaries to improve all future test runs through this component. We are currently investigating how to best use and deploy such techniques in production mode, including a centralized repository of test summaries for large parts of the Windows operating system.

5 Related Work

Blackbox fuzz testing in clusters at a large scale is not new. Nagy described a custom-built cluster dedicated to high volume testing of Microsoft Word that processes 1.7 million test cases per day [22]. The Office team at Microsoft has built a distributed fuzzing framework that works across "nights and weekends" use of idle desktops, as well as in clusters [14]. Google's security team devoted $2,000$ cores over roughly four weeks to fuzz testing Adobe Flash [11]. We leverage previous work in this area on the classification of crashing test cases, prioritization of bugs, and automatic reporting of important bugs to developers.

What is new is the use of whitebox fuzzing at the scale described in this paper. Whitebox fuzzing combines and extends program analysis, testing, verification, model checking and automated theorem proving techniques that have been developed over many years. One of the earliest proposals for using static analysis as a kind of symbolic program testing method was proposed by King almost 35 years ago [19]. The idea is to symbolically explore the tree of all computations the program exhibits when all possible value assignments to input parameters are considered. *Static test generation* consists of analyzing a program statically, by using symbolic execution techniques to attempt to compute inputs to drive the program along specific execution paths or branches, *without ever executing the program*. Unfortunately, this approach is ineffective whenever the program contains statements involving constraints outside the scope of reasoning of the theorem prover, i.e., statements "that cannot be reasoned about symbolically." This is frequent in practice due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions.

A recent breakthrough was the emergence of a second approach: *dynamic test generation* [16] [5]. It consists of executing the program, typically starting with some random inputs, while performing symbolic execution *dynamically*, collecting symbolic 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. A key advantage of dynamic test generation is that imprecision in symbolic execution can be alleviated using concrete values and randomization: whenever symbolic execution does not know how to generate a constraint for a program statement or library depending on some inputs, one can always simplify this constraint using the concrete values of those inputs [16].

Over the last few years, dynamic symbolic execution and test generation have been implemented in many tools, such as APOLLO, CUTE, KLEE, PEX and S2E (see [6] for a recent survey). For instance, the KLEE tool built on LLVM showed these techniques could yield high coverage on all programs in the `coreutils` and `busybox` suites, outperforming hand-generated tests created over multiple years. Follow-on tools have applied these techniques to testing drivers [7], and finding "trigger behavior" in malware [2]. While promising, this prior work did not report of any daily use of these techniques and tools.

Another recent significant milestone is the emergence of *whitebox fuzzing* [18] as the current main *"killer app"* for dynamic test generation, and arguably for automatic code-driven test generation in general. This in turn has allowed the developmemt of the next step in this decades-long journey: the first *"productization"* of large-scale symbolic execution and constraint solving. This paper is the first to report on this development.

Our work extends previous dynamic test generation tools with logging and control mechanisms. These mechanisms allow us to run whitebox fuzz testing for weeks on end with low effort and cost, and they enable data-driven improvements to our fuzzing platform. The closest related prior works are Metafuzz [21] and Cloud9 [3]. Metafuzz also performed logging of whitebox fuzzing on Linux, using the SmartFuzz plugin for Valgrind [21]. Unlike our current work, however, Metafuzz has no automatic control features. Furthermore, the authors do not show how data from Metafuzz directly inform future research or operations of whitebox fuzzing. Finally, the Metafuzz authors report experiments of 24 hours in length, while we report on multiple multi-week runs of whitebox fuzzing on many more applications.

Cloud9 [3] is a system for scaling KLEE-style symbolic execution across a cluster of commodity machines; the authors use Amazon's Elastic Compute Cloud in their experiments. Like KLEE, Cloud9 focuses on creating high coverage test suites for commodity programs. Their techniques for scaling symbolic execution across multiple machines are complementary to ours and could be used to inform the task partitioning we described in Section 3. Again, however, they do not report on multiple multi-week runs against hundreds of different test programs. Our work reports on usage and deployments which are orders of magnitude larger in all five scalability dimensions identified in Section 1 than any prior work in this space.

6 Conclusion

We have shown that whitebox fuzzing scales to production use. Our biggest pain points have been around the heterogeneity of different applications with respect to configuration and in the logistics of provisioning hundreds of machines. The systems we described here were developed in direct response to these pain points. The logging from SAGAN helps us keep track of hundreds of different programs and seed files. The JobCenter service then lets us turn around and deploy new configurations to machines running with our infrastructure.

Moving forward, we know from our data that significant challenges remain in improving the precision of symbolic execution and combating path explosion. As described in Section 2 and 4, we currently use unsound optimizations to scale to long execution traces and programs such as the Office suite, but those may also miss important bugs. Our tool also supports many optional features, such as reasoning with symbolic pointers [4, 10], which are sometimes expensive and therefore not always all turned on in production runs. By adding other monitoring features to SAGAN , we hope to drill down and understand better these cost/precision trade-offs.

We also recognize that whitebox fuzzing is only one piece of the security puzzle and one niche application for automatic test generation. The major longer-term win from all this technology comes in a shared infrastructure for the entire picture of people, processes, and tools required to build secure reliable software. Our infrastructure reduces the cost to "enroll" a program in this infrastructure, which is the gating factor to applying our techniques or any others. While in our case the infrastructure is on-premise, this could also involve the use of machines "in the cloud." No matter where it is located, we see centralized, data-driven security, testing and software engineering infrastructure as a key direction for future research.

7 Acklowledgments

This paper would not have been possible without the help and support of many people. We thank Michael Levin for starting the development of the SAGE tool and participating in this project for several years. We also thank Chris Marsh, Lei Fang, Stuart de Jong, and others in the former Center for Software Excellence for building the dynamic program analysis stack SAGE is built upon. We thank the developers of the Z3 SMT solver, including Nikolaj Bjorner and Leonardo de Moura. We appreciate the support of our managers in pursuing this project, including Tom Ball, Wolfram Schulte, and Helen Wang. Mehdi Bouaziz made substantial implementation contributions to JobCenter while an intern at Microsoft. Additional interns who have worked on SAGE include Dennis Jeffries, Adam Kiezun, Bassem Elkarablieh, Marius Nita, Cindy Rubio-Gonzalez, Johannes Kinder, Daniel Luchaup, and Louis Jachiet.

The work reported in this paper was performed in collaboration with our partners in Windows and in Office. In windows, we thank specifically Nick Bartmon, Eric Douglas, Dustin Duran, David Grant, Elmar Langholz, Vince Orgovan, Isaac Sheldon, Evan Tice, Dave Weston, and Russell West, for running SAGE and supporting development of the binary analysis tools on which it depends. In Office, we are particularly grateful to Tom Gallagher, Eric Jarvi and Octavian Timofte, for running SAGE and working with us to develop JobCenter. From the Microsoft Security Engineering Center, we thank Matt Miller, Lars Opstad, Andy Renk, Jason Shirk, and Dazhi Zhang. We thank our MSR colleagues Rich Draves, Jon Howell, Stuart Schecter, and Helen Wang for feedback on drafts of this paper under short notice. Finally, we thank all the users of SAGE across Microsoft.

References

[1] 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.

[2] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, New York, NY, USA, 2011. ACM.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

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

[6] C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N.Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In *ICSE'2011*, Honolulu, May 2011.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[8] Microsoft Corporation. Security development lifecycle, 2012. `http://www.microsoft.com/security/sdl/default.aspx`.

[9] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, April 2008. Springer-Verlag.

[10] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, pages 129–140, 2009.

[11] Ch. Evans, M. Moore, and T. Ormandy. Fuzzing at scale, 2011. `http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html`.

[12] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Trans. Netw.*, 2(2):122–136, April 1994.

[13] 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.

[14] T. Gallagher and D. Conger. Under the kimono of office security engineering. In *CanSecWest*, 2010.

[15] 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.

[16] 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.

[17] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In *Proceedings of SAS'2011 (18th International Static Analysis Symposium)*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128, Venice, September 2011. Springer-Verlag.

[18] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.

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

[20] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), December 1990.

[21] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, 2009.

[22] B. Nagy. Finding microsoft vulnerabilities by fuzzing binary files with ruby - a new fuzzing framework. In *SyScan*, 2009. `http://www.youtube.com/watch?v=u--j4YY_7cg`.

[23] 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.

[24] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.