

The Next Generation of Static Analysis

Boolean Satisfiability and
Path Simulation—A Perfect Match

Ben Chelf, Coverity CTO
Andy Chou, Coverity Chief Scientist

Introduction

Since its introduction, static source code analysis has had a mixed reputation with development teams due to long analysis times, excessive noise or an unacceptable rate of false-positive results. Excessive false-positive results are the main reason why many source code analysis products quickly become shelfware after a few uses. Despite early shortcomings, the promise of static analysis remained of interest to developers because the technology offers the ability to find bugs before software is run, improving code quality and dramatically accelerating the availability of new applications. Though static analysis has historically struggled to deliver on this promise, a groundbreaking new use of Boolean satisfiability (SAT) in the field is poised to help static analysis deliver on its potential.

This white paper will provide a brief overview of the history of static analysis and explain how the use of SAT in static analysis is enabling developers to improve the quality and security of their code by identifying a greater number of critical defects in their code with the lowest rate of false-positive results in the industry.

A Brief History of Static Analysis

The Need for Tools to Find Code Defects

Before the first software application was released, the first software defect(s) had been found and eliminated. Call them bugs, errors, failures or other names not suitable for publication – software defects have existed as long as software itself. As early applications evolved to become more robust and more complex, the remaining defects became more difficult to corral. Simply stated, the more lines of code necessary to create an application, the more defects one would expect to encounter during development.

Consider a theoretical average ratio of one defect per 1,000 lines of code (likely a gross underestimation). As a code base expands from thousands of lines, to tens of thousands, to hundreds of thousands, this defect ratio would become overwhelming for developers relying exclusively on manual techniques to control the resulting volume of defects.

With applications assuming more critical functions for business and industry, the consequence of defects in the field now mandates that software meet specific quality standards prior to release. It is at this intersection of opportunity and risk that developers turned to software itself to try and eliminate software defects earlier in the development life-cycle. Applying static analysis to software, the automated review of code prior to run-time with the intention of identifying defects, was an obvious solution to this fundamental challenge of ensuring code quality.

1st Generation Static Analysis

The first static analysis tool appeared in the late 70's. Known most commonly as Lint, this can be regarded as the 1st generation of commercially viable static analysis. Lint held great promise for developers when it was initially released. For the first time, developers had the ability to automate the detection of software defects very early in the application lifecycle, when they were easiest to correct. By extension, this would provide developers with greater confidence in the quality of their code prior to release. The technology 3 Boolean Satisfiability and Path Simulation—A Perfect Match behind Lint was revolutionary, because it used compilers to conduct defect checking, enabling it to become the first viable static source code analysis solution.

“As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.”

—Maurice Wilkes,
Inventor of the EDSAC,
1949

In reality though, Lint was not truly designed with the goal of identifying defects that cause run-time problems. Rather, its purpose was to flag suspicious or non-portable constructs in the code to help developers code in a more consistent format. By ‘suspicious code’ we mean code that, while technically correct from the perspective of the source code language (e.g., C, C++), might be structured so that it could possibly execute in ways that the developer did not intend. The problem with flagging suspicious code is, like compiler warnings, that code of this type could, and often would, work correctly. Because of this, and Lint’s limited analysis capabilities, noise rates were extremely high, often exceeding a 10:1 ratio between noise and real defects.

Consequently, finding genuine defects required developers to conduct time-consuming manual reviews of Lint’s results, compounding the exact problem that static analysis was supposed to eliminate. For this reason, Lint was never widely adopted as a defect detection tool, although it did enjoy some limited success with a few organizations. In fact, as a testament to the quality of Lint’s underlying technology, many different versions of the product still remain available today.

2nd Generation Static Analysis

For nearly two decades static analysis remained more fiction than fact as a commercially viable production tool for identifying defects. In early 2000 a second generation of tools (e.g., Stanford Checker) emerged that offered enough value to become commercially viable. By leveraging new technology that expanded the capabilities of first generation tools past simple pattern matching to also focus on path coverage, second generation static analysis was able to uncover more defects with real run-time implications.

These tools could also analyze entire code bases, not just one file. By shifting focus from ‘suspicious constructs’ to ‘run-time defects,’ developers of these new static analysis technologies recognized the need to understand more of the inter-workings of code bases. This meant combining sophisticated path analysis with inter-procedural analysis to understand what happened when the flow of control passed from one function to another within a given software system.

False Positives Vs. Noise in Static Analysis

There are two types of inaccurate results that can cause problems for static analysis solution—noise and false positives. A noisy result is one where the analysis is technically correct but the result is irrelevant. A false positive result is one where the analysis asserts a fact about the code (e.g., pointer could be null at line 150 and is dereferenced) and that fact is incorrect (e.g., the pointer could not possibly be null). It’s important for any static analysis to have a low rate of noise and false positives. In other words, it should be right most of the time (low false positive rate) and report results that the end users find to be relevant (low noise rate).

Despite their adoption and use by organizations, 2nd generation static analysis still had difficulty finding the sweet spot between accuracy and scalability. Some solutions were accurate for a small set of defect types, but could not scale to analyze millions of lines of code. Others could run in a short amount of time, but had accuracy rates similar to Lint, introducing the familiar false-positive and noise problems. When implemented, these tools can report defects at an acceptable ratio, but only with restricted analysis parameters.

The problem of wrestling with an elusive sweet spot between accuracy and scalability led to a false-positive problem that, like the noise problem preventing 1st generation tools from delivering on the promise of static analysis, slowed 2nd generation tools from being more rapidly adopted. While 2nd generation tools moved static analysis forward from a technology standpoint by identifying meaningful defects, their results were often found wanting for greater accuracy by developers.

Many 2nd generation tools suffered from the heterogeneity of build and development environments. Due to the inconsistency in development environments between organizations, 2nd generation tools could often require painful, time-consuming customization or integration efforts for teams that used anything but the most common technologies such as plain Makefiles with a single target, antfiles, or Visual Studio® project files.

3rd Generation Static Analysis

Today, a new generation of static analysis is emerging, technology that delivers unmatched quality of results from a solution that fits within existing development processes and environments. Leveraging solvers for Boolean satisfiability (SAT) to complement traditional path simulation analysis techniques, 3rd generation static analysis is providing developers with unmatched results that are both comprehensive and accurate.

This paper will explain the technology behind SAT in depth, but for now, let's define SAT as the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to 'true.' It is equally important to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically 'false' for

False Positives and Noise and 1st Generation Tools

False-positive and noisy results have always been a challenge for static analysis. Even the developers behind the most popular 1st generation tools recognized this. Consider a quote from Stephen Johnson, the inventor of Lint: "Lint tries to give information with a high degree of relevance. Messages of the form 'xxx might be a bug' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages."

—Stephen Johnson 7/26/1978

all possible variable assignments. In this latter case, we would say that the formula is unsatisfiable; otherwise it is satisfiable.

The innovative use of SAT allows static source code analysis to find the right defects in code without a high rate of distracting false positives. It also provides new capabilities that lay the foundation for further technological advances in static analysis. Before delving into the technology, some background on the use of SAT may provide some helpful context.

Boolean Satisfiability:

In complexity theory, the Boolean satisfiability problem (SAT) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: Given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true? A formula of propositional logic is said to be satisfiable if logical values can be assigned to its variables in a way that makes the formula true.

Leveraging SAT in Chip Design

In the field of computer-aided engineering and design, EDA (Electronic Design Automation) tools have been used with tremendous success to eliminate costly design failures in hardware devices. For decades companies such as Cadence, Mentor Graphics and Synopsys have used advanced tools to simulate the operations of circuits prior to fabrication to verify their correctness and performance.

The use of SAT was introduced in these tools in the 1980s to create a digital simulation of hardware, allowing developers to test a design's architectural operation and inspect the accuracy of performance at both cycle and interface levels. This visibility accelerated development and improved quality of chips by allowing EDA developers to more thoroughly simulate their designs prior to fabrication.

Because of its successful history in hardware development, the introduction of SAT to software analysis is unlike previous advancements in static analysis. Leveraging SAT solvers for performing static analysis of hardware designs is a mature, sophisticated technique that incorporates decades of refinement from our peers in the hardware tools industry. Coverity's patent-pending incorporation of this technology lays the groundwork for a new generation of static analysis, where we can analyze code with two complementary analysis techniques (SAT and Path Simulation) to deliver unmatched accuracy of results.

This paper will explain how SAT is pushing the technological capabilities of static analysis today, and how SAT offers the prospect of even greater innovation tomorrow.

Static Analysis Today

The 3rd Generation Exposed

The software development challenges of today require companies to look for innovative ways to remove the critical performance flaws and security vulnerabilities in software before it gets to the quality assurance lab, or worse, into the field. Fortunately for developers, source code analysis is now up to the task. By combining breakthrough techniques in the application of Boolean satisfiability to software with the latest advances in path simulation analysis, the most sophisticated source code analysis can boast out-of-the-box false positive rates as low as 10 percent and scalability to tens of millions of lines of C/C++ or Java code.

Software developers can use static analysis to automatically uncover errors that are typically missed by unit testing, system testing, quality assurance, and manual code reviews. By quickly finding and fixing these hard-to-find defects at the earliest stage in the software development life cycle, organizations are saving millions of dollars in associated costs.

The Price of Failure in Software

A 2002 National Institute of Standards and Technology (NIST) study titled “The Economic Impacts of Inadequate Infrastructure for Software Testing” estimated that software errors cost the U.S. economy an estimated \$59.5 billion annually. The report states that leveraging test infrastructures that allow developers to identify and fix defects earlier and more effectively could eliminate more than one-third of these costs. For more information on this study, visit http://www.nist.gov/public_affairs/releases/n02-10.htm.

Software DNA Map

Foundation of Superior Code Analysis

If developers want superior analysis, they must have a perfect picture of their software, because code analysis is only as accurate as the data it is based upon.

Imagine someone gave a software development team a system of a few million lines of code—a system they'd been working on for a long time—and asked them to draw a map of the software system. To be useful, this map would need to address all the following details: how every single file was compiled for all targets, how each set of files was linked together, the different binaries that were generated, the functions within each file and the corresponding callgraph, all the different control flow graphs through each function, and on and on. If the developers attempted to do this by hand, it would be practically impossible.

Today, automating this onerous task is possible, and the process of creating such a picture, known as a Software DNA Map, is the foundation that permits static analysis to significantly improve code quality and security by leveraging both path simulation analysis and SAT.

Software DNA Map

An extremely accurate representation of a software system based on understanding all operations that the build system performs as well as an authentic compilation of every source file in that build system. By providing an accurate representation of an application to the function, statement, and even expression level, the Software DNA Map enables static code analysis to overcome its previous limitations of excessive false positives and deliver accurate results that developers can put to immediate use.

Boolean Satisfiability (SAT)

As we discussed, the concept of Boolean satisfiability is not a new one, but recently, efficient SAT solvers have been developed to solve very complicated formulas with millions of variables.

A SAT solver is a computer program that takes in a formula of variables under the operations of AND, OR, and NOT and determines if there is a mapping of each individual variable to TRUE and FALSE such that the entire formula evaluates to TRUE (satisfiable). If there is at least one such mapping, the solver returns a specific satisfying assignment. If no such assignment exists, the SAT solver indicates that the formula is unsatisfiable and may provide a proof demonstrating that it is unsatisfiable.

The application of SAT to software requires that source code be represented in a form that can be plugged automatically into a SAT solver. Fortunately, with a Software DNA Map, all the necessary information from the code is available to transform it into any representation desired. Because SAT solvers deal in TRUE, FALSE, AND, OR, and NOT, the relevant portions of this program can be transformed into these constructs. Take an 8-bit variable as an example:

```
char a;
```

To represent 'a' as TRUES and FALSES, those 8 bits (1s and 0s) can be each thought of as TRUES and FALSES, so 'a' becomes an array of 8 Boolean values:

```
a0, a1, a2, a3, a4, a5, a6, a7
```

The operations that make up expressions in the code also must be translated. All expressions in code can be converted to an equivalent formula of AND, OR, and NOT. The thinking behind this is that a compiler must turn these operations into instructions in machine code and that machine code must run on a chip. The chip's circuitry is nothing more than pushing 1s and 0s (high voltage and low voltage) through a number of gates (all of which can be simplified to AND, OR, and NOT); therefore, this indicates that such a mapping exists. For example, to convert the expression:

```
a == 19
```

into a formula, the following expression would do the trick:

$$!a_7 \wedge !a_6 \wedge !a_5 \wedge a_4 \wedge !a_3 \wedge !a_2 \wedge a_1 \wedge a_0$$

In this example, a_0 is the low bit of 'a' and a_7 is the high bit. Plugging this into a SAT Solver would render the following assignment of variables for the formula to be satisfied:

$a_0 = \text{True (1)}$

$a_1 = \text{True (1)}$

$a_2 = \text{False (0)}$

$a_3 = \text{False (0)}$

$a_4 = \text{True (1)}$

$a_5 = \text{False (0)}$

$a_6 = \text{False (0)}$

$a_7 = \text{False (0)}$

Taking that, as binary 00010011, shows that it is equivalent to 19.

Once the entire Software DNA Map is represented in this format of TRUES, FALSES, NOTS, ANDS, and ORS, a wide variety of formulas can be constructed from this representation and SAT solvers can be applied to analyze the code for additional, more sophisticated quality and security problems. It is this bit-accurate representation of the software that enables more precise static analysis than previously was possible based solely on path simulation.

Bit-accurate Software Representation

A representation of a software system in terms of formulas that are made up of Boolean variables under the logical operators AND, OR, and NOT.

With a bit-accurate representation, static analysis can now leverage SAT solvers to analyze the software in this additional way as a complement to path simulation techniques. The diagrams below show the difference between a traditional path simulation representation (Figure 1) and bit-accurate representation (Figure 2):

Figure 1: Path Simulation

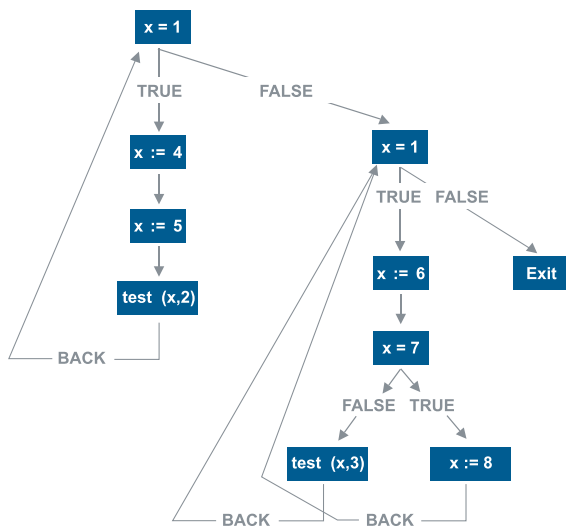
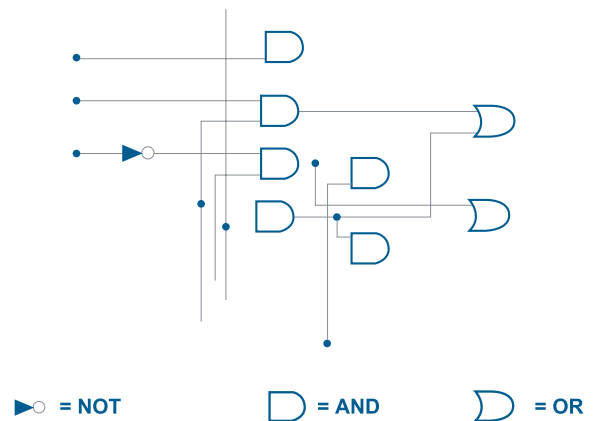


Figure 2: Bit-Accurate



Once we have a bit-accurate representation of the code, the next step is to determine what questions we pose to SAT solvers to obtain useful information from them. Equally important is an understanding of how these questions interact with existing path simulation analysis to improve the overall quality of our analysis.

SAT: False Path Pruning

The first application of SAT solvers in static analysis to date has been for false path pruning. When performing path simulation analysis, sometimes a defect will be reported on a path that is infeasible at runtime. This reported defect should be eliminated from the static analysis results because there is no possible combination of values of variables where that defect could actually occur in the application. The methodologies for dealing with this problem within a path simulation analysis framework pale in comparison to the abilities a bit-accurate representation and a good SAT solver can provide in this regard.

For example, with a bit-accurate representation of the source code, a 3rd generation static analysis solution can construct a formula that is a conjunction (AND) of all the conditions in a path that lead to any given defect discovered by path simulation analysis. By solving this formula, the SAT solver can indicate if there is a set of values for the variables involved in all the conditions such that the path can actually be executed. If the SAT solver says “satisfiable,” then the path is feasible at runtime. If the SAT solver says “unsatisfiable,” the path can never be executed and no bug should be reported.

Scalability in SAT and Path Simulation Analysis

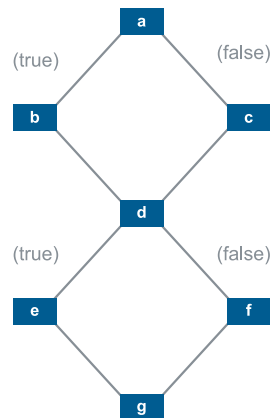
With the technological heft of combined path simulation analysis and SAT-solving, many readers might be wondering if it's possible to perform all of this analysis yet still handle large amounts of code. The foundational technology in a Software DNA Map permits 3rd generation static analysis to effectively combine Path Simulation and SAT based analysis while scaling to millions of lines of code. The power of adding SAT to Path Simulation comes from the fact that the strengths of SAT allow the analysis to compensate for the natural weaknesses that occur when traditional Path Simulation is made to scale over millions of lines of code.

In path simulation analysis, in order to scale to millions of lines of code in a reasonable amount of time, the code must be abstracted to a higher level representation that consists of states and transitions between those states. For example, when performing analysis that looks for NULL pointer dereferences, the analysis need not care about the some 4 billion different possible values for every pointer (assuming a 32 bit world). The analysis only cares about two values: NULL and NOT NULL.

As you can see, this simplifying assumption significantly reduces the state space that must be explored. One drawback of this technique is that it also gives up some precision of value since now we consider all pointers that are NOT NULL to have the same value, but if we are only looking for places where NULL pointers are dereferenced, this is not a problem in practice.

With reduced state space, we must also look for ways to reduce the complexity of the total number of paths explored through the code base. Because there are an exponential number of paths through any given software system, tracing all those paths would lead to an analysis solution that is incapable of effectively scaling. Coverity's patent-pending technology meets this challenge by aggressively caching states that have already been explored. Going back to our pointer example, imagine we had a control flow graph that looked like the following:

Figure 3: Control Flow Graph



There are clearly four paths through this code base (a-b-d-e-g, a-c-d-e-g, a-b-d-f-g, a-c-d-f-g). However, we may not need to explore all four paths in the code. If there are no pointer operations in any of the blocks a-g, the analysis can do the following:

- 1) Explore the path a-b-d-e-g. (NOTE: There are no pointers, so no state changes occur)
- 2) Back track to the last decision point (d)
- 3) Explore f-g (hence, a-b-d-f-g). (NOTE: There are no pointers, so no state changes occur)
- 4) Then backtrack to the first decision point (a). Explore the other paths that begin with a-c-d

It is when we reach point (d) in this particular case that we employ the use of cached knowledge of states to save analysis time. In exploring the third path, once we reach (d) for the third time, we know that all of the paths below (d) have been exhausted, and since we are entering (d) with the same state that we previously entered it, we can stop analyzing paths because we know the analysis has already covered all possible cases. Obviously, this is a very simplified view of software, but hopefully it illustrates the general concept behind caching states when performing path analysis.

All reasonable 2nd generation static analysis tools must perform some sort of this abstraction and caching in order to scale to millions of lines of code and still report reasonable defects. However, as you can imagine, the abstractions can lead to simplifying assumptions which in turn can lead to false positives or false negatives. With incorrect simplifying assumptions, caching may 'cache out' in the wrong places. Because of the abstraction and caching that path simulation analysis already performs, SAT-based techniques need to be incorporated in this caching to scale properly.

Combining SAT-based and Path Simulation Analysis

An area where SAT differentiates 2nd and 3rd generation static analysis is in determining which paths in your code should be explored. One challenge that has historically limited the effectiveness of path simulation-only techniques is the elimination of paths that should be explored (improperly deciding a path does not need to be explored) and the exploration of paths that should not be explored (improperly deciding that a path could actually execute at run time when in fact it cannot). To give an example of the latter, consider the control flow graph above (Figure 3), but with the decisions at blocks (a) and (d) being the following:

[a]: if (x == 0)

[d]: if (x != 0)

Assuming x does not change in blocks (b) or (c), while there might appear to be 4 paths through the control flow graph, we know that because of the dependency between the condition of (a) and condition of (d), there are only 2 paths through the code base. If the analysis decides to explore the path a-b-d-e-g, this would be a FALSE path because it's impossible to execute at runtime. Moreover, if the analysis reported a defect on this path, that defect would clearly be a false positive since that path cannot exist when running the program.

To tackle the false path problem with SAT, we rely upon path simulation analysis to explore the paths, but before reporting a defect, the analysis conjoins all the conditions that must be true on a given path and runs the resulting formula through a SAT solver. If the SAT solver reports that the formula is satisfiable, we know the path must be feasible and the defect is reported. However, if SAT reports that it is unsatisfiable, we know we have explored a false path and the defect should not be reported. Given the above example, the formula we would plug into a SAT solver would be:

x == 0 AND x != 0

Clearly this is not satisfiable for any values of x (x can't be both zero and not zero) so we would know to prune this path and not report a defect, even if our path simulation analysis did not have the right abstraction to capture this data.

Adding SAT to path simulation analysis in this way does lead to a few technical challenges. Because performing a SAT computation does introduce some system overhead, we must discriminate when to leverage SAT solvers in order to maintain scalability with large code bases. Ideally, false paths are pruned as soon as there is an inconsistency in the conditions that lead to that path.

Consider the above example with $x == 0$ and $x != 0$, but now imagine that the function is much longer and these two decisions are the first two in the function. Using traditional path simulation techniques, we may explore what is essentially the same false path multiple times due to the caching behavior of any given check that we perform. If that occurs, then SAT may be called a tremendous number of times to compute essentially the same thing (that x can't be zero and non-zero).

To avoid this problem, whenever we perform a false path computation using SAT and it proves to be unsatisfiable (a false path), we reduce the formula to the simplest set of conditions that lead to the formula being unsatisfiable (a known technique in SAT-solving) and then annotate those nodes in the control flow graph for later use with the path simulation engine. This way, we essentially leverage the concept of caching not only with the path simulation states, but also with the information that the SAT solver returns when it performs a path computation.

This interaction between path simulation analysis and SAT-based techniques is at the crux of Coverity's patent-pending use of SAT in static analysis, without which there would be no way for SAT to scale beyond a few thousand lines of code.

The True-Positive Result

Consider for a moment our earlier review of the history of static analysis and the problems with excessive false positive results. By identifying false paths with SAT, developers are no longer tasked with sifting through incorrect results from their static analysis products. This enables them to focus testing and analysis efforts on potential problems that have a real possibility of compromising the project at hand.

Early test results of leveraging SAT-based false path pruning in the field have been very promising. Based on data from more than 10 open source projects at <http://www.scan.coverity.com>, a joint venture between Coverity, Symantec and the U.S. Department of Homeland Security, false-positive rates dropped by an additional 15% when the false path pruning SAT solver was incorporated in the overall analysis.

Future Uses of SAT in Static Analysis

It's important to note that false path pruning is just one example of how SAT can be leveraged to deliver better static source code analysis. Other potential applications of SAT include the ability to find problems such as string overflows, integer overflows or deadcode, and the use of static assert checking to identify difficult-to-find logic bugs. While some instances of the defects in these categories can be discovered today, SAT-based analysis will allow us to build on the success of existing path simulation analysis to reach new levels of accuracy and precision in static code analysis.

SAT will benefit software developers and architects by fundamentally changing the way they view static analysis of their source code. Just as going from simple “grep” parsing through code to practical path simulation analysis earlier this decade was a huge eye-opener for many developers (“You mean, it can show me the whole path to get to a real bug?”), leveraging SAT solvers for static analysis can impress anyone who writes software (“How could you possibly know that about my code?”). Because of this, SAT is the technological breakthrough that introduces a new generation of static analysis, bringing us one step closer to delivering on its long-awaited promise.

Author Biographies

Ben Chelf is CTO of San Francisco-based Coverity, Inc. Before he co-founded Coverity, Ben was a founding member of the Stanford Computer Science Laboratory team that architected and developed Coverity's underlying technology. He is one of the world's leading experts on the commercial use of static source code analysis. In 2007, he was selected as one of Computerworld's "40 Innovative IT People to Watch, Under the Age of 40," and often provides expert insight into software quality and security for the press, public audiences, and in published writings. In his role at Coverity, Mr. Chelf works with organizations such as the US Department of Homeland Security, Cisco, Ericsson and Symantec to improve the security and quality of software around the world. He holds BS and MS degrees in Computer Science from Stanford University.

Andy Chou is Chief Scientist of San Francisco-based Coverity Inc. Andy is responsible for advancing source code analysis technology at Coverity as well as furthering the state-of-the-art in software quality and security industry wide. Prior to co-founding Coverity, Andy was instrumental in developing the core intellectual property behind the Coverity platform. He received his Ph.D. in Computer Science from Stanford University and his B.S. in Electrical Engineering from UC Berkeley.

About Coverity

Coverity (www.coverity.com), the leader in improving software quality and security, is a privately held company headquartered in San Francisco. Coverity's groundbreaking technology enables developers to control complexity in the development process by automatically finding and helping to repair critical software defects and security vulnerabilities throughout the application lifecycle. More than 450 leading companies including ARM, Phillips, RIM, Rockwell-Collins, Samsung and UBS rely on Coverity to help them ensure the delivery of superior software.

Free Trial

Request a free Coverity trial and see first hand how to rapidly detect and remediate serious defects and vulnerabilities. No changes to your code are necessary. There are no limitations on code size, and you will receive a complimentary report detailing actionable analysis results. Register for the on-site evaluation at www.coverity.com or call us at (800) 873-8193.

Headquarters

185 Berry Street, Suite 2400
San Francisco, CA 94107
(800) 873-8193
<http://www.coverity.com>
sales@coverity.com

Boston

230 Congress Street
Suite 303
Boston, MA 02110
(617) 933-6500

UK

Coverity Limited
Magdalen Centre
Robert Robinson Avenue
The Oxford Science Park
Oxford OX4 4GA
England

Japan

Coverity Asia Pacific
Level 32, Shinjuku Nomura Bldg.
1-26-2 Nishi-Shinjuku, Shinjuku-ku
Tokyo 163-0532
Japan