SETH HALLEM, DAVID PARK, DAWSON ENGLER, COVERITY

Uprooting Software Defects at the Source

Source code analysis is an emerging technology in the software industry that allows critical source code defects to be detected before a program runs. Although the concept of detecting programming errors at compile time is not new, the technology to build effective tools that can process millions of lines of code *and* report substantive defects with only a small amount of noise has long eluded the market. At the same time, a different type of solution is needed to combat current trends in the software industry that are steadily diminishing the effectiveness of conventional software testing and quality assurance. These trends include:

Uprooting Software Defects at the Source

1. Increasing CPU performance and decreasing memory costs, which enable the development of larger, more complex software.
2. The emergence of the Internet, which creates an unpredictable execution environment for most software and attaches a security risk to every software defect.
3. Increased demands for software quality fueled by the growth of embedded systems and software services.

Source code analysis addresses these trends by effectively searching for violations of programming rules down all executable paths through the source code. The types of rules amenable to automated analysis encompass a wide variety of properties that programmers have long known and have long struggled to obey without any help from development tools.

SOFTWARE QUALITY CRISIS

A May 2002 report prepared for the National Institute of Standards and Technologies (NIST)¹ estimates the annual cost of software defects in the United States as \$59.5 billion. Each defect that goes undetected until after a product has shipped can cost software producers many tens of thousands of dollars to address and patch. The cost suffered by the users of that product is often orders of magnitude higher.

The root cause of these astronomical costs is the increasing complexity of today's software. In the early days of computing, there were strict memory limits that inherently kept complexity in check by limiting code size. As these memory requirements have disappeared and processor performance has improved, the requirements for software have fundamentally changed. Modern server-side applications such as operating systems, application servers, and databases usually contain hundreds of thousands, if not millions, of lines of source code.

An unfortunate repercussion of today's world of networked computing contributes a second factor to the economic impact of software quality. Once software is operating in a networked environment, virtually every bug in that software becomes a potential security hole. If there is any way for an outsider to trigger the particular code path to the bug, that bug is now at least a DoS (Denial of

Service) attack waiting to be discovered, if not worse. The occurrences of such security attacks have grown exponentially over the past several years, and more than eight out of ten corporations in the United States were victims of security breaches during the past year.²

While the movement to safe languages (e.g., Java, .Net) has alleviated some of the challenges in building secure software, these environments are still subject to another insidious type of security vulnerability: incorrect outsider validation, which allows unintended access to private information. This problem is often traced to errors in the source code that an analysis tool could detect. As Java systems are deployed at banks, insurance companies, and government agencies, the effects of improper information access could become very costly.

A third trend that has fundamentally increased the economic impact of software quality is the proliferation of embedded devices. Software that runs inside a device generally carries the expectation of continuous uptime. Due to these expectations, embedded devices have a prohibitively high software patch cost. In addition to embedded devices, the spread of connectivity through the Internet and corporate intranets has extended these high patch costs to every file server, software router, service provider, and more, that is expected to operate continuously on the network. A single mainframe computer is no longer the only costly point of failure.

The software industry is responding to these trends by bringing reliability to the foreground. First, redundancy has emerged as an immediate, though expensive, work-around to maintaining reliability. Many popular Web sites are fueled by hundreds or thousands of interchangeable machines that can be rebooted without significantly influencing the overall traffic patterns. In some cases, machines are rebooted intentionally after a short period of time to avoid pathological performance problems (i.e., thrashing) caused by memory leaks in the software.

Second, reliability guarantees are becoming critical indicators of competitive differentiation. Vendors including Sun, Hewlett-Packard, and IBM are promising high availability for their infrastructure products, while Oracle delivers an "unbreakable" promise.³ These place substantial pressure on those vendors to deliver higher-quality software and, in the event of software failures, to develop effective workarounds in a short period of time.

THE NEED FOR AUTOMATED SOURCE CODE ANALYSIS

The 2002 NIST report⁴ estimates that "feasible" improvements to testing infrastructures could reduce the annual

cost of software defects in the United States by \$22.2 billion. The gap between the feasible improvement and the total cost of \$59.5 billion reflects the well-known fact that no solution can find all of the bugs in any substantial software project. The \$22.2 billion reduction is an estimate based on the economical viability of current research and development trends in the software quality industry.

The specific improvements cited in NIST's report fall into two categories: (1) detecting errors earlier in the software development cycle, closer to the points where they are introduced, and (2) reducing the cost of fixing defects by locating the root causes of bugs faster and with more precision.⁵ Given this cost structure, it is not difficult to see the value of automated source code analysis. Source code analysis can identify defects at compile time as soon as the code is written. In addition, because the detected defects are derived from the code itself, source code analyzers can pinpoint the location and root cause of each error.

In contrast, devoting additional resources to traditional testing will lead to diminishing returns and will ultimately fail to keep pace with the growing size and complexity of software. The best practices for software testing employ a toolset to both manage and monitor a large suite of tests that are intended to exercise software in real-world situations. Management tools include test generators (mostly available for Web applications) and test case-management tools. Monitoring solutions include code coverage tools, which determine what percentage of the code is exercised by a test suite, and dynamic analysis tools that instrument the code to pinpoint the causes of test failures (e.g., Purify, Insure++, BoundsChecker). While these tools enhance the efficiency of software testing, they do not address the fundamental challenge that arises from the path explosion property.

The path explosion property states that the number of executable paths through a program scales exponentially with the number of lines of code. Even if 100 percent

code coverage were achieved during testing, it might still amount to a small coverage of the executable paths through the code. For most large commercial software, it is infeasible to account for all of the scenarios through which the software will run. Indeed, even if all of the test scenarios were somehow magically enumerated, it would generally take the lifetime of the universe to manually execute and test every feasible path in a complex program.

It is important to note that there will never be a technique to entirely replace traditional testing. It is impossible to completely automate tests for such software features as user interaction and business logic. However, there are large classes of critical defects that can be detected automatically at compile time and go beyond the syntactic errors that traditional compilers catch.

Furthermore, when a property is amenable to source code analysis, the analysis will almost always check that the property is not violated more effectively than a human can determine with a manual code review. Because the analysis is done at compile time rather than at runtime, it can search for defects along all paths through the code. Many of these paths are both difficult to exercise with conventional runtime testing and difficult for humans to follow during a code review. Adding source code analysis to the development process can drastically reduce a costly class of bugs that were previously detected either during integration testing or in the field. By eliminating these bugs on the developer's desktop, the testing organization can focus entirely on those properties that are beyond the scope of an automated tool.

Toy Code Example Illustrating the 0-Address Property for Pointers

FIG 1

```
int test(int x)
{
    int *pointer;
    if (x == 1) {
        pointer = 0; /* A */
    } else {
        pointer = (int*) malloc(sizeof *pointer); /* B */
    }

    if (!pointer) {
        printf("Bad pointer. Should exit!"); /* C */
    } else {
        *pointer = 10;
    }
    return *pointer; /* D */
}
```

Uprooting Software Defects at the Source

ANALYZING THE SOURCE CODE

The basic idea of source code analysis is to simulate the execution of the code within a significantly simplified model of the execution environment. For the purposes of this discussion, a *state* is a set of values that reflects the current status of a running program. The state includes all of the accessible values in memory, all of the relevant files on the hard disk, and the next instruction in the instruction queue. The *state space* is defined as the set of all possible states. Clearly, enumerating the set of all possible values that could be on the hard disk, in memory, and in the instruction queue will produce an effectively infinite list. This observation is one possible justification for the path explosion property.

The key insight for source code analysis, however, is the fact that, for any particular program property, not all of the data that comprises a state is relevant. Consider the example function written in the C programming language in figure 1. While our example function may accept an arbitrary integer, a value of 1 indicates that the pointer *p* is set to the 0 address, while any other value indicates that the same pointer is initialized to a block of accessible memory on the heap (assuming that the call to *malloc* can never fail). In general, an attempt to access

pointers containing the value 0 as the address will cause the operating system to terminate the program.

If we are analyzing the source code to detect illegal accesses of this type, we only care about two possible states for each pointer in the program: the pointer is 0 (the *null* state), and the pointer is not 0 (the *not-null* state). Reducing the state space to these possibilities effectively reduces the possible values in our state space from billions of values (32 bits) to two values (1 bit). Rather than evaluating each statement in the program by making the appropriate modifications to memory, the hard disk, and so forth, we simply focus on the pointer values along each executable code path. In figure 1, there are two executable paths, one that executes statements A, C, and D, and one that executes statements B and D. Statement A indicates to our analysis that “pointer” is subsequently 0 along the current code path. At statement D, the analysis recognizes an illegal de-reference of a pointer with value 0, and an error is reported. Along the alternative path, statement B assigns a valid, de-referenceable value to “pointer,” and we do not report an error in this function.

While the example above may seem too simple to exist in production code, the example in figure 2 taken directly from the Linux kernel illustrates that even these simple classes of defects are prevalent in widely deployed code bases (see figure 2). While this type of error is often detected during integration testing, a source code analysis tool could have detected the error before the erroneous code was introduced into the code repository.

Even within this simplified state space, it is not feasible to accurately determine in all cases whether a pointer can hold a 0 value. Thoroughly reasoning about

FIG 2

Code Fragment from the File “drivers/net/pppoe.c” in the Linux 2.5.54 Development Kernel

```
...
if (!po) {
    int hash = hash_item(po->pppoe_pa.sid, po->pppoe_pa.remote);
    ...
}
...
```

This code fragment shows how even the simplest type of NULL pointer mistakes are common amongst experienced programmers. If the variable *po* is 0, the expressions *po->pppoe_pa.sid* and *po->pppoe_pa.remote* will both crash the kernel when executed. If released, this bug would affect all Linux computers that communicate over consumer DSL lines using PPPoE. This bug has been fixed in the latest release snapshot of the Linux kernel.

complex arithmetic operations or non-trivial data structures at compile time is theoretically undecidable. Thus, any source code analysis tool must make approximations.

Academic research has traditionally focused on complete verification. In this context, a verifier would make the assumption that whenever it is unclear whether a pointer has value 0, it is safest to assume that it does have the value 0 and report errors accordingly. Analyses that make this assumption often produce false positive ratios as high as 100 or even 1,000 false reports per true bug report. The alternative assumption is that whenever the analysis cannot be sure that a pointer is 0, it is best to assume that the pointer is not 0. This assumption may cause the analysis to miss certain classes of defects, but it produces a significantly more usable tool. In our experience, a tool with a good heuristic approach can still detect a high percentage of the errors.

DEFINING THE SPACE

In general, almost any property that can be articulated in terms of the source code can be converted into an analysis that searches for violations of that property. Prototypical examples include:

Memory leaks: Each pointer has an associated state *allocated*, which indicates that the pointer refers to an allocated block of memory that must be returned to the system, or *unallocated*, which means that the pointer does not point to an allocated block of memory.

File handle leaks: Each file handle has an attached state *opened*, which indicates that the handle refers to an open file, or *unopened*, which indicates that the handle does not point to an open file.

Permissions checks: Certain operations must be guarded by a permissions check along all executable paths, otherwise a program is vulnerable to unauthorized access. A source code analysis can check that the permissions of the calling process are correctly validated before a protected operation is executed. The state of the system is a global value indicating the permissions level along the current code path. The state space is simply the list of all possible permissions levels.

Buffer overruns: Each buffer has an attached state that is the allocated size of that buffer. Each index variable has an attached state that is the value of that index variable.

Once the state space is simplified, software bugs can be reduced to property violations specified in terms of the abstracted space. For instance, a memory-leak violation occurs when a locally defined pointer goes into the *allocated* state down some code path and fails to revert to the *unallocated* state before the enclosing routine exits.

In general, the easiest way to determine if a property is amenable to source code analysis is to start by identifying a code fragment that violates that property. Either a bug that was diagnosed in an existing product or a toy example will suffice. If it is possible to complete this step, there is usually some type of analysis that can be done to automatically detect violations of the identified form.

The next step is to determine how intelligent the analysis will have to be to produce good results. In general, the more knowledge the analysis must have of the data structures in the code or the input values to the program, the more difficult it will be to track a property precisely. In this case, there is generally an effective heuristic that can be applied to identify a particular subset of the errors. Using multiple heuristic analyses often produces better results than a single, precise analysis.

For example, to determine if violations of property 3 are amenable to source code analysis, we might start with a known security hole. The security hole is probably identified in the code by a call to an operating system (OS) function without a surrounding condition that validates the permissions of the executing program. At this point, we could construct a simple analysis that looks for calls to the OS function in question without any enclosing conditional check. That analysis could be further refined by enumerating how the source code for a correct permissions check would be written, then enhancing the analysis to report errors when there is an enclosing conditional check but that check is incorrect. As a further refinement, if there are several OS functions that should be executed only by a privileged user, we could enumerate a mapping from check to function and encode that mapping directly into our analysis.

The properties described above indicate that the value of source code analysis applies across all programming languages. While some of the specific problems (e.g., memory leaks) do not apply to safe languages, all programs interact with resources and must obey programming interfaces that encode rules that can be checked by a source code analysis tool. Thus, the value of these tools is not limited to any particular programming language or even to a particular programming paradigm.

EXPLORING THE SPACE

Once the state space is defined, the next task is to search the program state space for violations of each property. The basic techniques for searching the state space are derived from a body of research referred to as data-flow analysis.

The basic idea in data-flow analysis is to search the

Uprooting Software Defects at the Source

state space of the entire program until no new information can be derived (fixed point). The program is first represented as a graph, where each node is an executable statement from the source code and the edges in the graph indicate the flow of control in the program. A violation, such as a memory leak, can be described as an associated state machine (see figure 3) that tracks the state of each pointer along each path through the program graph. At each node in the program graph, a pointer can be in the *allocated* or *unallocated* state. Nodes in the program graph (or statements) may trigger transitions in this state machine when they contain program constructs that are relevant to the property being checked. For example, a call to `malloc`, a memory allocation function in the standard C library, can trigger a state-machine transition from *unallocated* to *allocated*, and a return statement, or any other statement that exits a scope, can trigger a transition from *allocated* to *error*.

The program graph is thus searched in a depth-first fashion, monitoring changes to the state machine. A key insight is that the search can be dramatically optimized if we keep track of the state-machine states we have seen at each program graph node. Even though there are usually multiple paths to a particular node, we do not have to revisit nodes that we have seen before, as long as the tracking state machine was also in the same state at those nodes during a previous visit. Revisiting such nodes is unnecessary because all of the possible differences in the state of the running program between the two visits are irrelevant to the property being checked.

By using this optimization, the analysis is able to avoid the path-explosion problem because the performance of the algorithm now scales linearly with the size of the program, rather than exponentially. This type of data-flow analysis has worked well in practice, discovering thousands of critical bugs and exploitable security holes in the Linux and OpenBSD operating systems.⁶

A downside of the approach just described is that it requires the manual specification of each program property that is checked by the analysis. One solution is to combine a source code analysis algorithm with a data-mining algorithm that attempts to infer correct behavior

directly from the source code in order to alleviate some of the specification burden. Although these automated inference techniques are an important feature in making source code analyses effective as a product, a complete explanation of the techniques is outside the scope of this discussion.

AVAILABLE TOOLS

The idea of source code analysis has existed in tool form for many years through a family of tools derived from Lint.⁷ While Lint has spawned variants in both academia and industry, the original focus of Lint was to provide additional rigor to C source code by essentially imposing a C style guide on all programmers. Attempts to create Lint-like tools with more extensive error-checking capabilities have generally failed. The fundamental shortcoming with these Lint variants has always been the ratio of errors reported to actual flaws detected. These tools often report tens or even hundreds of messages that may reflect bad programming style without reflecting a real defect in the analyzed software. The reported errors may also simply reflect shortcomings in the power of the tool. In addition, the effectiveness of the tool is closely tied to the addition of specifications to the actual source code, thereby requiring developers within an organization to adopt a disciplined use of the Lint annotations.

The next generation of source code analysis tools is placing all of the required specifications inside the tool, allowing the tool to churn through millions of lines of source code without a single modification to the code. In addition, these tools include a much more sophisticated model of the program state, allowing them to discover more complex classes of software defects and achieve false error rates that, in the worst cases, are closer to 20 percent. As these tools are integrated into commercial development processes, those rates will dip even further because programmers will recognize coding styles and idioms that are amenable to tool analysis and naturally gravitate toward them.

The goal is to make source code analysis an industry best practice, just as the automated test suite has been for the past ten years. There was a time when a few hand-written tests were generally sufficient to validate that a program was working as intended. There also was a time when researchers believed that it would be possible to prove that a program was working correctly. Though those methodologies are no longer viable, providing a tool that acts as a layer of insulation against programmer errors can enable a company to prevent hundreds, if not thousands, of critical software defects from reaching production. Q

REFERENCES

1. Tassey, G. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002: see <http://www.nist.gov/director/prog-ofc/report02-3.pdf> (membership required).
2. Cisco Systems. *Economic Impact of Network Security Threats*. Cisco White Paper. Dec. 2002: see http://www.cisco.com/warp/public/cc/so/neso/sqso/roi1_wp.pdf.
3. Oracle: see <http://www.oracle.com/oramag/oracle/02-mar/index.html?o22break.html>.
4. Tassey, G. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002: see <http://www.nist.gov/director/prog-ofc/report02-3.pdf> (membership required).
5. Tassey, G. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. Prepared by RTI for the National Institute of Standards and Technology (NIST), May 2002: see <http://www.nist.gov/director/prog-ofc/report02-3.pdf> (membership required).
6. Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions, *Proceedings of Operating Systems Design and Implementation* (Sept. 2002): see also <http://metacomp.stanford.edu/osdi2000/paper.html>.
7. Johnson, S. C. Lint, a C program checker. *Unix Programmer's Manual*, 1978: see <http://plan9.bell-labs.com/7thEdMan/vol2/lint>.

LOVE IT, HATE IT? LET US KNOW:

feedback@acmqueue.com or www.acmqueue.com/forums

SETH HALLEM is a cofounder and senior architect at Coverity, a Menlo Park, California-based start-up specializing in source code analysis tools. In less than one year, Coverity has grown from an idea in a Stanford laboratory to a company whose customers range from start-ups to the Fortune 100. Prior to Coverity, Hallem was a Ph.D. candidate at Stanford University, working with Dawson Engler on the metacompilation project. During that time, he coauthored several academic publications in respected conferences in the fields of operating systems and programming languages. Hallem's work is currently focused on designing and developing source code analysis tools for the Java programming language.

DAVID PARK is a cofounder of Coverity and manages business development and sales efforts at the fast-growing start-up. As an early contributor to a number of venture-funded start-ups in e-business, telecom, and networking, Park has helped bring successful ideas to market for the past several years. Prior to that, he was a Ph.D. candidate in computer science at Stanford University, where he worked on tools to model check networking protocols and concurrent Java programs. He is the coauthor of several publications in software specification and verification, and is the recipient of a National Science Foundation Fellowship and the Stanford Terman Engineering Award.

DAWSON ENGLER is an assistant professor at Stanford University and a cofounder of Coverity. His research interests are operating systems and compilers. His work at Stanford University has focused on software checking in both the static analysis and model checking fields. Engler's metacompilation project has generated more than ten publications in respected academic journals, and one fast-growing start-up (Coverity). Prior to joining the faculty at Stanford University, Engler received a Ph.D. from the Massachusetts Institute of Technology, where he cofounded the exokernel operating systems project.

© 2003 ACM 1542-7730/03/1100 \$5.00.