DOI:10.1145/1646353.1646374

How Coverity built a bug-finding tool, and a business, around the unlimited supply of bugs in software systems.

BY AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPEAK, AND DAWSON ENGLER

A Few Billion Lines of Code Later

Using Static Analysis to Find Bugs in the Real World

IN 2002, COVERITY commercialized[3] a research static bug-finding tool.[6,9] Not surprisingly, as academics, our view of commercial realities was not perfectly accurate. However, the problems we encountered were not the obvious ones. Discussions with tool researchers and system builders suggest we were not alone in our naïveté. Here, we document some of the more important examples of what we learned developing and commercializing an industrial-strength bug-finding tool.

We built our tool to find generic errors (such as memory corruption and data races) and system-specific or interface-specific violations (such as violations of function-ordering constraints). The tool, like all static bug finders, leveraged the fact that programming rules often map clearly to source code; thus static inspection can find many of their violations. For example, to check the rule "acquired locks must be released," a checker would look for relevant operations (such as `lock()` and `unlock()`) and inspect the code path after flagging rule disobedience (such as `lock()` with no `unlock()` and double locking).

For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).

A glance through the literature reveals many ways to go about static bug finding.[1,2,4,7,8,11] For us, the central religion was results: If it worked, it was good, and if not, not. The ideal: check millions of lines of code with little manual setup and find the maximum number of serious true errors with the minimum number of false reports. As much as possible, we avoided using annotations or specifications to reduce manual labor.

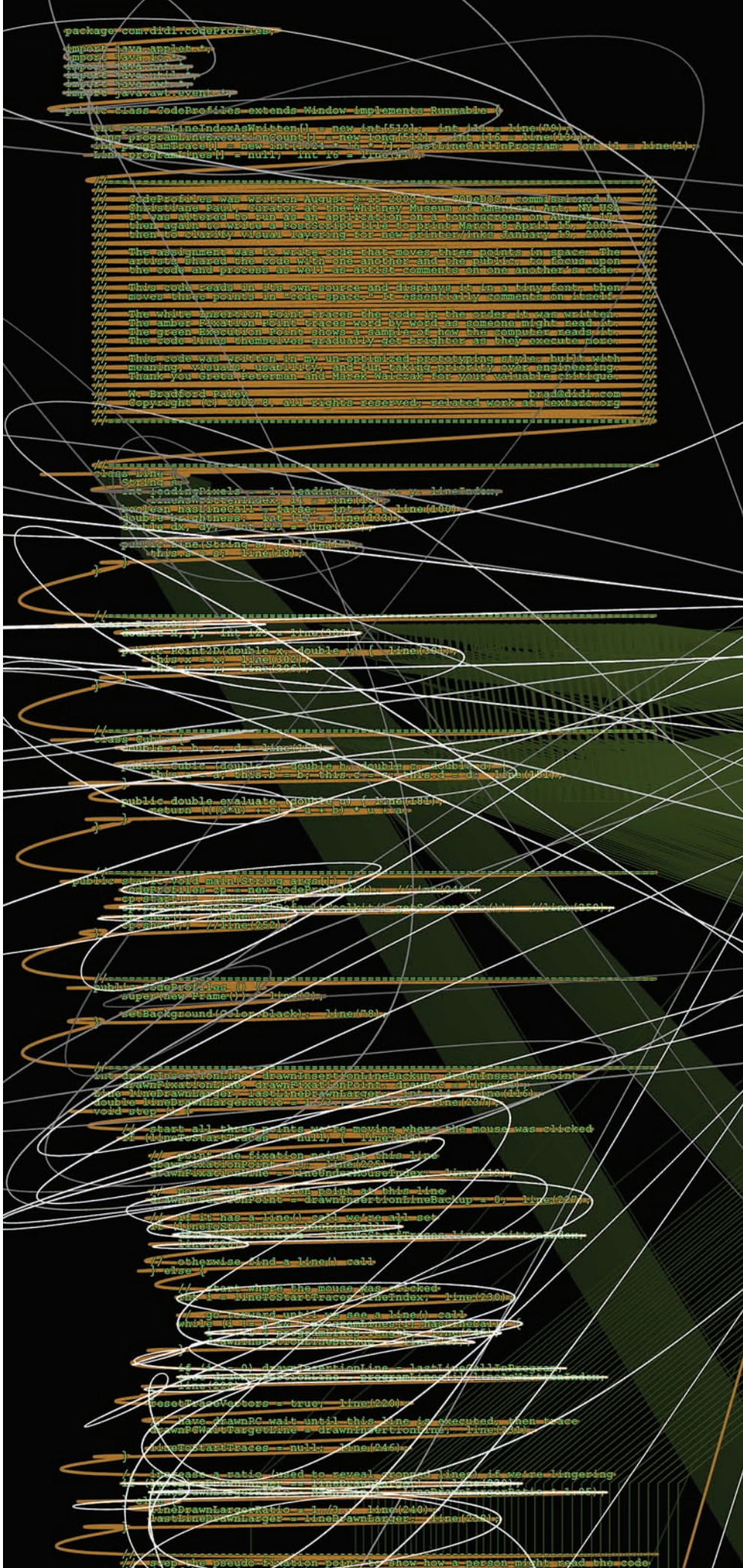
Like the PREfix product,[2] we were also unsound. Our product did not verify the absence of errors but rather tried to find as many of them as possible. Unsoundness let us focus on handling the easiest cases first, scaling up as it proved useful. We could ignore code constructs that led to high rates of false-error messages (false positives) or analysis complexity, in the extreme skipping problematic code entirely (such as assembly statements, functions, or even entire files). Circa 2000, unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects.

Initially, publishing was the main force driving tool development. We would generally devise a set of checkers or analysis tricks, run them over a few

million lines of code (typically Linux), count the bugs, and write everything up. Like other early static-tool researchers, we benefited from what seems an empirical law: Assuming you have a reasonable tool, if you run it over a large, previously unchecked system, you will always find bugs. If you don't, the immediate knee-jerk reaction is that something must be wrong. Misconfiguration? Mistake with macros? Wrong compilation target? If programmers must obey a rule hundreds of times, then without an automatic safety net they cannot avoid mistakes. Thus, even our initial effort with primitive analysis found hundreds of errors.

This is the research context. We now describe the commercial context. Our rough view of the technical challenges of commercialization was that given that the tool would regularly handle "large amounts" of "real" code, we needed only a pretty box; the rest was a business issue. This view was naïve. While we include many examples of unexpected obstacles here, they devolve mainly from consequences of two main dynamics:

First, in the research lab a few people check a few code bases; in reality many check many. The problems that show up when thousands of programmers use a tool to check hundreds (or even thousands) of code bases do not show up when you and your co-authors check only a few. The result of summing many independent random variables? A Gaussian distribution, most of it not on the points you saw and adapted to in the lab. Furthermore, Gaussian distributions have tails. As the number of samples grows, so, too, does the absolute number of points several standard deviations from the mean. The unusual starts to occur with increasing frequency.



W. Bradford Paley's CodeProfiles was originally commissioned for the Whitney Museum of American Art's "CODeDOC" Exhibition and later included in MoMA's "Design and the Elastic Mind" exhibition. CodeProfiles explores the space of code itself; the program reads its source into memory, traces three points as they once moved through that space, then prints itself on the page.

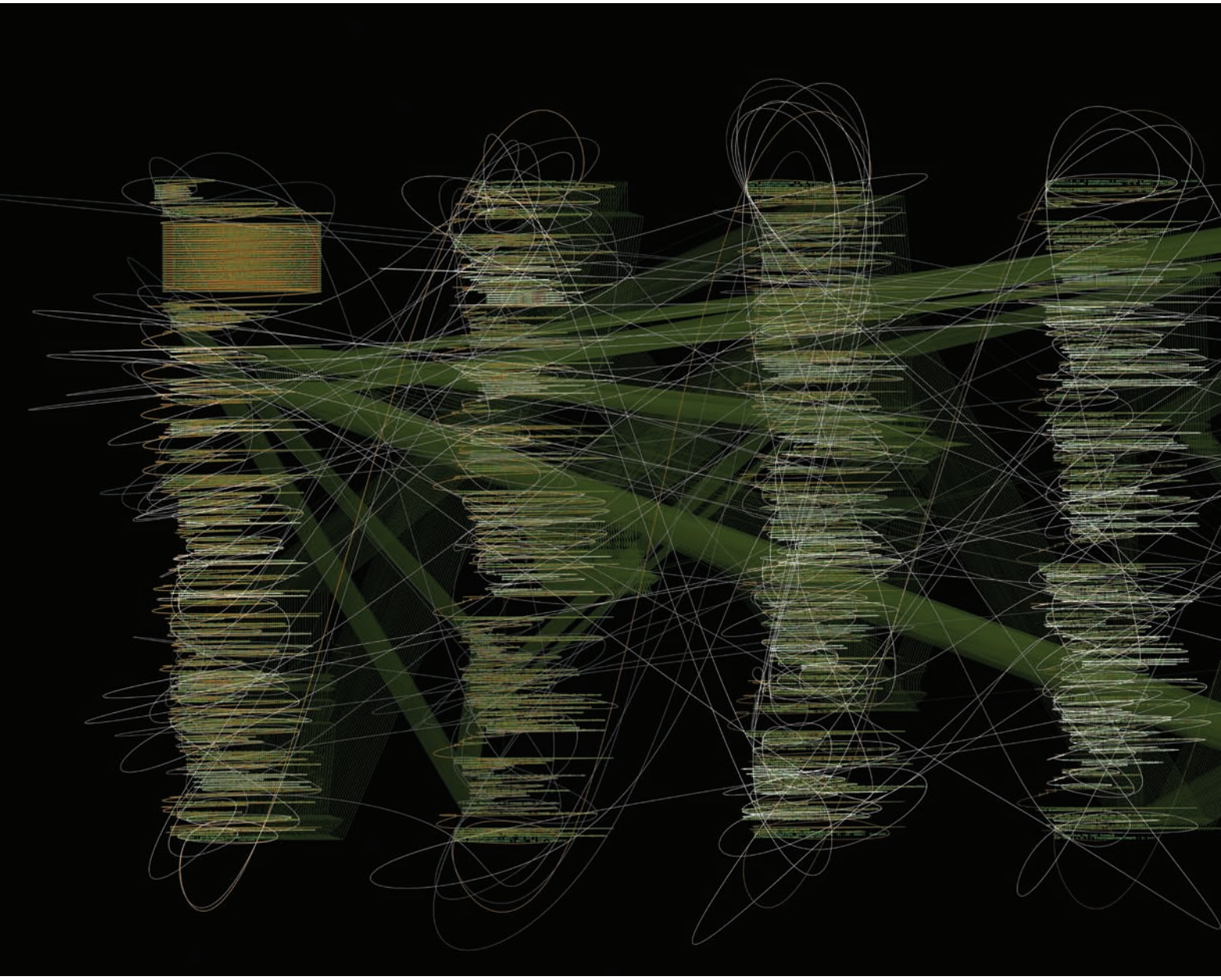
"CODEPROFILES" BY W. BRADFORD PALEY. HTTP://DIDI.COM

For code, these features include problematic idioms, the types of false positives encountered, the distance of a dialect from a language standard, and the way the build works. For developers, variations appear in raw ability, knowledge, the amount they care about bugs, false positives, and the types of both. A given company won't deviate in all these features but, given the number of features to choose from, often includes at least one weird oddity. Weird is not good. Tools want expected. Expected you can tune a tool to handle; surprise interacts badly with tuning assumptions.

Second, in the lab the user's values, knowledge, and incentives are those of the tool builder, since the user and the builder are the same person. Deployment leads to severe fission; users often have little understanding of the tool and little interest in helping develop it (for reasons ranging from simple skepticism to perverse reward incentives) and typically label any error message they find confusing as false. A tool that works well under these constraints looks very different from one tool builders design for themselves.

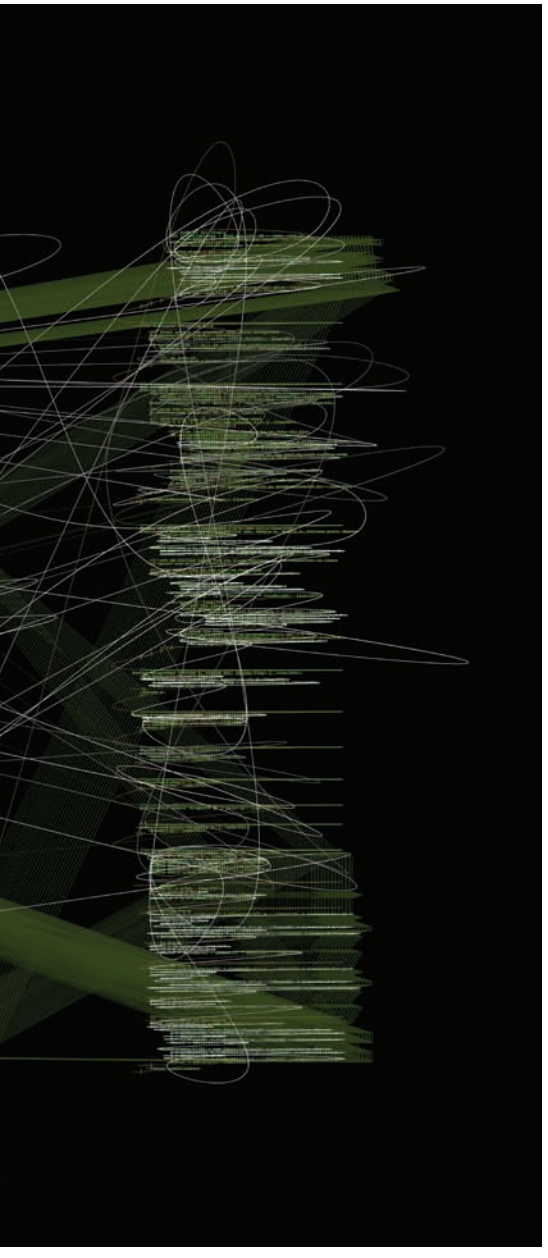
However, for every user who lacks the understanding or motivation one might hope for, another is eager to understand how it all works (or perhaps already does), willing to help even beyond what one might consider reasonable. Such champions make sales as easily as their antithesis blocks them. However, since their main requirements tend to be technical (the tool must work) the reader likely sees how to make them happy, so we rarely discuss them here.

Most of our lessons come from two different styles of use: the initial trial of the tool and how the company uses the tool after buying it. The trial is a pre-sale demonstration that attempts to show that the tool works well on a potential customer's code. We generally ship a salesperson and an engineer to the customer's site. The engineer configures the tool and runs it over a given code base and presents results soon after. Initially, the checking run would happen



in the morning, and the results meeting would follow in the afternoon; as code size at trials grows it's not uncommon to split them across two (or more) days.

Sending people to a trial dramatically raises the incremental cost of each sale. However, it gives the non-trivial benefit of letting us educate customers (so they do not label serious, true bugs as false positives) and do real-time, ad hoc workarounds of weird customer system setups.



"CODEPROFILES" BY W. BRADFORD PALEY

The trial structure is a harsh test for any tool, and there is little time. The checked system is large (millions of lines of code, with 20–30MLOC a possibility). The code and its build system are both difficult to understand. However, the tool must routinely go from never seeing the system previously to getting good bugs in a few hours. Since we present results almost immediately after the checking run, the bugs must be good with few false positives; there is no time to cherry pick them.

Furthermore, the error messages must be clear enough that the sales engineer (who didn't build the checked system or the tool) can diagnose and explain them in real time in response to "What about this one?" questions.

The most common usage model for the product has companies run it as part of their nightly build. Thus, most require that checking runs complete in 12 hours, though those with larger code bases (10+MLOC) grudgingly accept 24 hours. A tool that cannot analyze at least 1,400 lines of code per minute makes it difficult to meet these targets. During a checking run, error messages are put in a database for subsequent triaging, where users label them as true errors or false positives. We spend significant effort designing the system so these labels are automatically reapplied if the error message they refer to comes up on subsequent runs, despite code-dilating edits or analysis-changing bug-fixes to checkers.

As of this writing (December 2009), approximately 700 customers have licensed the Coverity Static Analysis product, with somewhat more than a billion lines of code among them. We estimate that since its creation the tool has analyzed several billion lines of code, some more difficult than others.

Caveats. Drawing lessons from a single data point has obvious problems. Our product's requirements roughly form a "least common denominator" set needed by any tool that uses nontrivial analysis to check large amounts of code across many organizations; the tool must find and parse the code, and users must be able to understand error messages. Further, there are many ways to handle the problems we have encountered, and our way may not be the best one. We discuss our methods more for specificity than as a claim of solution.

Finally, while we have had success as a static-tools company, these are small steps. We are tiny compared to mature technology companies. Here, too, we have tried to limit our discussion to conditions likely to be true in a larger setting.

Laws of Bug Finding

The fundamental law of bug finding is No Check = No Bug. If the tool can't check a system, file, code path, or given property, then it won't find bugs in it. Assuming a reasonable tool, the first order bound on bug counts is just how much code can be shoved through the tool. Ten times more code is 10 times more bugs.

We imagined this law was as simple a statement of fact as we needed. Unfortunately, two seemingly vacuous corollaries place harsh first-order bounds on bug counts:

Law: You can't check code you don't see. It seems too trite to note that checking code requires first finding it... until you try to do so consistently on many large code bases. Probably the most reliable way to check a system is to grab its code during the build process; the build system knows exactly which files are included in the system and how to compile them. This seems like a simple task. Unfortunately, it's often difficult to understand what an ad hoc, homegrown build system is doing well enough to extract this information, a difficulty compounded by the near-universal absolute edict: "No, you can't touch that." By default, companies refuse to let an external force modify anything; you cannot modify their compiler path, their broken makefiles (if they have any), or in any way write or reconfigure anything other than your own temporary files. Which is fine, since if you need to modify it, you most likely won't understand it.

Further, for isolation, companies often insist on setting up a test machine for you to use. As a result, not infrequently the build you are given to check does not work in the first place, which you would get blamed for if you had touched anything.

Our approach in the initial months of commercialization in 2002 was a low-tech, read-only replay of the build commands: run `make`, record its output in a file, and rewrite the invocations to their compiler (such as gcc) to instead call our checking tool, then rerun everything. Easy and simple. This approach worked perfectly in the lab and for a small number of our earliest customers. We then had the fol-

lowing conversation with a potential customer:

"How do we run your tool?"

"Just type '`make`' and we'll rewrite its output."

"What's '`make`'? We use `ClearCase`."

"Uh, What's `ClearCase`?"

This turned out to be a chasm we couldn't cross. (Strictly speaking, the customer used '`ClearMake`,' but the superficial similarities in name are entirely unhelpful at the technical level.) We skipped that company and went to a few others. They exposed other problems with our method, which we papered over with 90% hacks. None seemed so troublesome as to force us to rethink the approach—at least until we got the following support call from a large customer:

"Why is it when I run your tool, I have to reinstall my Linux distribution from CD?"

This was indeed a puzzling question. Some poking around exposed the following chain of events: the company's `make` used a novel format to print out the absolute path of the directory in which the compiler ran; our script misparsed this path, producing the empty string that we gave as the destination to the Unix "`cd`" (change directory) command, causing it to change to the top level of the system; it ran "`rm -rf *`" (recursive delete) during compilation to clean up temporary files; and the build process ran as root. Summing these points produces the removal of all files on the system.

The right approach, which we have used for the past seven years, kicks off the build process and intercepts every system call it invokes. As a result, we can see everything needed for checking, including the exact executables invoked, their command lines, the directory they run in, and the version of the compiler (needed for compiler-bug workarounds). This control makes it easy to grab and precisely check all source code, to the extent of automatically changing the language dialect on a per-file basis.

To invoke our tool users need only call it with their build command as an argument:

```
cov-build <build command>
```

We thought this approach was bulletproof. Unfortunately, as the astute reader has noted, it requires a command prompt. Soon after implementing it we went to a large company, so large it had a hyperspecialized build engineer, who engaged in the following dialogue:

A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive.

"How do I run your tool?"

"Oh, it's easy. Just type '`cov-build`' before your build command."

"Build command? I just push this [GUI] button..."

Social vs. technical. The social restriction that you cannot change anything, no matter how broken it may be, forces ugly workarounds. A representative example is: Build interposition on Windows requires running the compiler in the debugger. Unfortunately, doing so causes a very popular windows C++ compiler—Visual Studio C++ .NET 2003—to prematurely exit with a bizarre error message. After some high-stress fussing, it turns out that the compiler has a use-after-free bug, hit when code used a Microsoft-specific C language extension (certain invocations of its `#using` directive). The compiler runs fine in normal use; when it reads the freed memory, the original contents are still there, so everything works. However, when run with the debugger, the compiler switches to using a "debug malloc," which on each `free` call sets the freed memory contents to a garbage value. The subsequent read returns this value, and the compiler blows up with a fatal error. The sufficiently perverse reader can no doubt guess the "solution."[a]

Law: You can't check code you can't parse. Checking code deeply requires understanding the code's semantics. The most basic requirement is that you parse it. Parsing is considered a solved problem. Unfortunately, this view is naïve, rooted in the widely believed myth that programming languages exist.

The C language does not exist; neither does Java, C++, and C#. While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler. Further, they equate compilation with certification. A file their compiler does

a Immediately after process startup our tool writes 0 to the memory location of the "in debugger" variable that the compiler checks to decide whether to use the debug `malloc`.

not reject has been certified as "C code" no matter how blatantly illegal its contents may be to a language scholar. Fed this illegal not-C code, a tool's C front-end will reject it. This problem is the tool's problem.

Compounding it (and others) the person responsible for running the tool is often not the one punished if the checked code breaks. (This person also often doesn't understand the checked code or how the tool works.) In particular, since our tool often runs as part of the nightly build, the build engineer managing this process is often in charge of ensuring the tool runs correctly. Many build engineers have a single concrete metric of success: that all tools terminate with successful exit codes. They see Coverity's tool as just another speed bump in the list of things they must get through. Guess how receptive they are to fixing code the "official" compiler accepted but the tool rejected with a parse error? This lack of interest generally extends to any aspect of the tool for which they are responsible.

Many (all?) compilers diverge from the standard. Compilers have bugs. Or are very old. Written by people who misunderstand the specification (not just for C++). Or have numerous extensions. The mere presence of these divergences causes the code they allow to appear. If a compiler accepts construct X, then given enough programmers and code, eventually X is typed, not rejected, then encased in the code base, where the static tool will, not helpfully, flag it as a parse error.

The tool can't simply ignore divergent code, since significant markets are awash in it. For example, one enormous software company once viewed conformance as a competitive disadvantage, since it would let others make tools usable in lieu of its own. Embedded software companies make great tool customers, given the bug aversion of their customers; users don't like it if their cars (or even their toasters) crash. Unfortunately, the space constraints in such systems and their tight coupling to hardware have led to an astonishing oeuvre of enthusiastically used compiler extensions.

Finally, in safety-critical software systems, changing the compiler often requires costly re-certification. Thus, we routinely see the use of decades-old compilers. While the languages these compilers accept have interesting features, strong concordance with a modern language standard is not one of them. Age begets new problems. Realistically, diagnosing a compiler's divergences requires having a copy of the compiler. How do you purchase a license for a compiler 20 versions old? Or whose company has gone out of business? Not through normal channels. We have literally resorted to buying copies off eBay.

This dynamic shows up in a softer way with non-safety-critical systems; the larger the code base, the more the sales force is rewarded for a sale, skewing sales toward such systems. Large code bases take a while to build and often get tied to the compiler used when they were born, skewing the average age of the compilers whose languages we must accept.

If divergence-induced parse errors are isolated events scattered here and there, then they don't matter. An unsound tool can skip them. Unfortunately, failure often isn't modular. In a sad, too-common story line, some crucial, purportedly "C" header file contains a blatantly illegal non-C construct. It gets included by all files. The no-longer-potential customer is treated to a constant stream of parse errors as your compiler rips through the customer's source files, rejecting each in turn. The customer's derisive stance is, "Deep source code analysis? Your tool can't even compile code. How can it find bugs?" It may find this event so amusing that it tells many friends.

Tiny set of bad snippets seen in header files. One of the first examples we encountered of illegal-construct-in-key-header file came up at a large networking company

```
// "redefinition of parameter 'a'"
void foo(int a, int a);
```

The programmer names `foo`'s first formal parameter a and, in a form of lexical locality, the second as well. Harmless. But any conformant compiler will reject this code. Our tool certainly did. This is not helpful; compiling no files means finding no bugs, and people don't need your tool for that. And, because its compiler accepted it, the potential customer blamed us.

Here's an opposite, less-harmless case where the programmer is trying to make two different things the same

```
typedef char int;
```

("Useless type name in empty declaration.")

And one where readability trumps the language spec

```
unsigned x = 0xdead _ beef;
```

("Invalid suffix '_beef' on integer constant.")

From the embedded space, creating a label that takes no space

```
void x;
```

("Storage size of 'x' is not known.")

Another embedded example that controls where the space comes from

```
unsigned x @ "text";
```

("Stray '@' in program.")

A more advanced case of a nonstandard construct is

```
Int16 ErrSetJump(ErrJumpBuf buf)
 = { 0x4E40 + 15, 0xA085; }
```

It treats the hexadecimal values of machine-code instructions as program source.

The award for most widely used extension should, perhaps, go to Microsoft support for precompiled headers. Among the most nettlesome troubles is that the compiler skips all the text before an inclusion of a precompiled header. The implication of this behavior is that the following code can be compiled without complaint:

```
I can put whatever I want here.
It doesn't have to compile.
If your compiler gives an error,
   it sucks.
#include  <some-precompiled-
   header.h>
```

Microsoft's on-the-fly header fabrication makes things worse.

Assembly is the most consistently troublesome construct. It's already non-portable, so compilers seem to almost deliberately use weird syntax, making it difficult to handle in a general way. Unfortunately, if a programmer uses assembly it's probably to write a widely used function, and if the programmer does it, the most likely place to put it is in a widely used

header file. Here are two ways (out of many) to issue a `mov` instruction

```
// First way
foo() {
      _ _asm mov eax, eab
      mov eax, eab;
}

// Second way
#pragma asm
_ _asm [ mov eax, eab mov
eax, eab ]
#pragma end _asm
```

The only thing shared in addition to mov is the lack of common textual keys that can be used to elide them.

We have thus far discussed only C, a simple language; C++ compilers diverge to an even worse degree, and we go to great lengths to support them. On the other hand, C# and Java have been easier, since we analyze the bytecode they compile to rather than their source.

How to parse not-C with a C front-end. OK, so programmers use extensions. How difficult is it to solve this problem? Coverity has a full-time team of some of its sharpest engineers to firefight this banal, technically uninteresting problem as their sole job. They're never done.[b]

We first tried to make the problem someone else's problem by using the Edison Design Group (EDG) C/C++ front-end to parse code.[5] EDG has worked on how to parse real C code since 1989 and is the de facto industry standard front-end. Anyone deciding to not build a homegrown front-end will almost certainly license from EDG. All those who do build a homegrown front-end will almost certainly wish they did license EDG after a few experiences with real code. EDG aims not just for mere feature compatibility but for version-specific bug compatibility across a range of compilers. Its front-end probably resides near the limit of what a profitable company can do in terms of front-end gyrations.

Unfortunately, the creativity of compiler writers means that despite two decades of work EDG still regularly meets defeat when trying to parse real-world large code bases.[c] Thus, our next step is for each supported compiler, we write a set of "transformers" that mangle its personal language into something closer to what EDG can parse. The most common transformation simply rips out the offending construct. As one measure of how much C does not exist, the table here counts the lines of transformer code needed to make the languages accepted by 18 widely used compilers look vaguely like C. A line of transformer code was almost always written only when we were burned to a degree that was difficult to work around. Adding each new compiler to our list of "supported" compilers almost always requires writing some kind of transformer. Unfortunately, we sometimes need a deeper view of semantics so are forced to hack EDG directly. This method is a last resort. Still, at last count (as of early 2009) there were more than 406(!) places in the front-end where we had an `#ifdef COVERITY` to handle a specific, unanticipated construct.

EDG is widely used as a compiler front-end. One might think that for customers using EDG-based compilers we would be in great shape. Unfortunately, this is not necessarily the case. Even ignoring the fact that compilers based on EDG often modify EDG in idiosyncratic ways, there is no single "EDG front-end" but rather many versions and possible configurations that often accept a slightly different language variant than the (often newer) version we use. As a Sisyphean twist, assume we cannot work around and report an incompatibility. If EDG then considers the problem important enough to fix, it will roll it together with other patches into a new version.

So, to get our own fix, we must upgrade the version we use, often causing divergence from other unupgraded EDG compiler front-ends, and more issues ensue.

Social versus technical. Can we get customer source code? Almost always, no. Despite nondisclosure agreements, even for parse errors and preprocessed code, though perhaps because we are viewed as too small to sue to recoup damages. As a result, our sales engineers must type problems in reports from memory. This works as well as you might expect. It's worse for performance problems, which often show up only in large-code settings. But one shouldn't complain, since classified systems make things even worse. Can we send someone onsite to look at the code? No. You listen to recited syntax on the phone.

Bugs

Do bugs matter? Companies buy bug-finding tools because they see bugs as bad. However, not everyone agrees that bugs matter. The following event has occurred during numerous trials. The tool finds a clear, ugly error (memory corruption or use-after-free) in important code, and the interaction with the customer goes like thus:

"So?"

"Isn't that bad? What happens if you hit it?"

"Oh, it'll crash. We'll get a call." [Shrug.]

If developers don't feel pain, they often don't care. Indifference can arise from lack of accountability; if QA cannot reproduce a bug, then there is no blame. Other times, it's just odd:

"Is this a bug?"

"I'm just the security guy."

"That's not a bug; it's in third-party code."

"A leak? Don't know. The author left years ago..."

No, your tool is broken; that is not a bug. Given enough code, any bug-

b Anecdotally, the dynamic memory-checking tool Purify[10] had an analogous struggle at the machine-code level, where Purify's developers expended significant resources reverse engineering the various activation-record layouts used by different compilers.

c Coverity won the dubious honor of being the single largest source of EDG bug reports after only three years of use.

Lines of code per transformer for 18 common compilers we support.

160 QNX	280 HP-UX	285 picc.cpp
294 sun.java.cpp	384 st.cpp	334 cosmic.cpp
421 intel.cpp	457 sun.cpp	603 iccmsa.cpp
629 bcc.cpp	673 diab.cpp	756 xlc.cpp
912 ARM	914 GNU	1294 Microsoft
1425 keil.cpp	1848 cw.cpp	1665 Metrowerks

finding tool will uncover some weird examples. Given enough coders, you'll see the same thing. The following utterances were culled from trial meetings:

Upon seeing an error report saying the following loop body was dead code

```
foo(i = 1; i < 0; i++)
  ... deadcode ...
```

"No, that's a false positive; a loop executes at least once."

For this memory corruption error (32-bit machine)

```
int a[2], b;
memset(a, 0, 12);
```

"No, I meant to do that; they are next to each other."

For this use-after-free

```
free(foo);
foo->bar = ...;
```

"No, that's OK; there is no malloc call between the free and use."

As a final example, a buffer overflow checker flagged a bunch of errors of the form

```
unsigned p[4];
...
p[4] = 1;
```

"No, ANSI lets you write 1 past the end of the array."

After heated argument, the programmer said, "We'll have to agree to disagree." We could agree about the disagreement, though we couldn't quite comprehend it. The (subtle?) interplay between 0-based offsets and buffer sizes seems to come up every few months.

While programmers are not often so egregiously mistaken, the general trend holds; a not-understood bug report is commonly labeled a false positive, rather than spurring the programmer to delve deeper. The result? We have completely abandoned some analyses that might generate difficult-to-understand reports.

How to handle cluelessness. You cannot often argue with people who are sufficiently confused about technical matters; they think you are the one who doesn't get it. They also tend to get emotional. Arguing reliably kills sales. What to do? One trick is to try to organize a large meeting so their peers do the work for you. The more people in the room, the more likely there is someone very smart and respected and cares (about bugs and about the given code), can diagnose an error (to counter arguments it's a false positive), has been burned by a similar error, loses his/her bonus for errors, or is in another group (another potential sale).

...it's not uncommon for tool improvement to be viewed as "bad" or at least a problem.

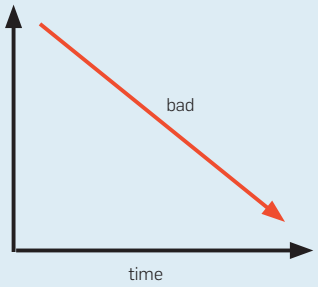
Further, a larger results meeting increases the probability that anyone laid off at a later date attended it and saw how your tool worked. True story: A networking company agreed to buy the Coverity product, and one week later laid off 110 people (not because of us). Good or bad? For the fired people it clearly wasn't a happy day. However, it had a surprising result for us at a business level; when these people were hired at other companies some suggested bringing the tool in for a trial, resulting in four sales.

What happens when you can't fix all the bugs? If you think bugs are bad enough to buy a bug-finding tool, you will fix them. Not quite. A rough heuristic is that fewer than 1,000 bugs, then fix them. More? The baseline is to record the current bugs, don't fix them but do fix any new bugs. Many companies have independently come up with this practice, which is more rational than it seems. Having a lot of bugs usually requires a lot of code. Much of it won't have changed in a long time. A reasonable, conservative heuristic is if you haven't touched code in years, don't modify it (even for a bug fix) to avoid causing any breakage.

A surprising consequence is it's not uncommon for tool improvement to be viewed as "bad" or at least a problem. Pretend you are a manager. For anything bad you can measure, you want it to diminish over time. This means you are improving something and get a bonus.

You may not understand technical issues that well, and your boss certainly doesn't understand them. Thus, you want a simple graph that looks like Figure 1; no manager gets a bonus for Figure 2. Representative story: At company X, version 2.4 of the tool found approximately 2,400 errors, and over time the company fixed about 1,200 of them. Then it upgraded to version 3.6. Suddenly there were 3,600 errors. The manager was furious for two reasons: One, we "undid" all the work his people

Figure 1. Bugs down over time = manager bonus.



had done, and two, how could we have missed them the first time?

How do upgrades happen when more bugs is no good? Companies independently settle on a small number of upgrade models:

Never. Guarantees "improvement";

Never before a release (where it would be most crucial). Counterintuitively happens most often in companies that believe the tool helps with release quality in that they use it to "gate" the release;

Never before a meeting. This is at least socially rational;

Upgrade, then roll back. Seems to happen at least once at large companies; and

Upgrade only checkers where they fix most errors. Common checkers include use-after-free, memory corruption, (sometimes) locking, and (sometimes) checkers that flag code contradictions.

Do missed errors matter? If people don't fix all the bugs, do missed errors (false negatives) matter? Of course not; they are invisible. Well, not always. Common cases: Potential customers intentionally introduced bugs into the system, asking "Why didn't you find it?" Many check if you find important past bugs. The easiest sale is to a group whose code you are checking that was horribly burned by a specific bug last week, and you find it. If you don't find it? No matter the hundreds of other bugs that may be the next important bug.

Here is an open secret known to bug finders: The set of bugs found by tool A is rarely a superset of another tool B, even if A is much better than B. Thus, the discussion gets pushed from "A is better than B" to "A finds some things, B finds some things" and does not help the case of A.

Adding bugs can be a problem; losing already inspected bugs is always a problem, even if you replace them with many more new errors. While users know in theory that the tool is "not a verifier," it's very different when the tool demonstrates this limitation, good and hard, by losing a few hundred known errors after an upgrade.

The easiest way to lose bugs is to add just one to your tool. A bug that causes false negatives is easy to miss. One such bug in how our early research tool's internal representation handled array references meant the analysis ignored most array uses for more than nine months. In our commercial product, blatant situations like this are prevented through detailed unit testing, but uncovering the effect of subtle bugs is still difficult because customer source code is complex and not available.

Churn

Users really want the same result from run to run. Even if they changed their code base. Even if they upgraded the tool. Their model of error messages? Compiler warnings. Classic determinism states: the same input + same function = same result. What users want: different input (modified code base) + different function (tool version) = same result. As a result, we find upgrades to be a constant headache. Analysis changes can easily cause the set of defects found to shift. The new-speak term we use internally is "churn." A big change from academia is that we spend considerable time and energy worrying about churn when modifying checkers. We try to cap churn at less than 5% per release. This goal means large classes of analysis tricks are disallowed since they cannot obviously guarantee minimal effect on the bugs found. Randomization is verboten, a tragedy given that it provides simple, elegant solutions to many of the exponential problems we encounter. Timeouts are also bad and sometimes used as a last resort but never encouraged.

Myth: More analysis is always good. While nondeterministic analysis might cause problems, it seems that adding more deterministic analysis is always good. Bring on path sensitivity! Theorem proving! SAT solvers! Unfortunately, no.

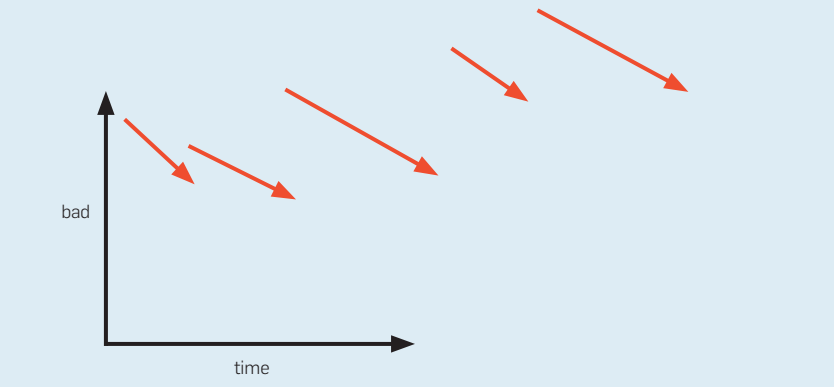
At the most basic level, errors found with little analysis are often better than errors found with deeper tricks. A good error is probable, a true error, easy to diagnose; best is difficult to misdiagnose. As the number of analysis steps increases, so, too, does the chance of analysis mistake, user confusion, or the perceived improbability of event sequence. No analysis equals no mistake.

Further, explaining errors is often more difficult than finding them. A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive. The heuristic we follow: Whenever a checker calls a complicated analysis subroutine, we have to explain what that routine did to the user, and the user will then have to (correctly) manually replicate that tricky thing in his/her head.

Sophisticated analysis is not easy to explain or redo manually. Compounding the problem, users often lack a strong grasp on how compilers work. A representative user quote is "'Static' analysis'? What's the performance overhead?"

The end result? Since the analysis that suppresses false positives is invisible (it removes error messages rather than generates them) its sophistication has scaled far beyond what our research

Figure 2. No bonus.



system did. On the other hand, the commercial Coverity product, despite its improvements, lags behind the research system in some ways because it had to drop checkers or techniques that demand too much sophistication on the part of the user. As an example, for many years we gave up on checkers that flagged concurrency errors; while finding such errors was not too difficult, explaining them to many users was. (The PREfix system also avoided reporting races for similar reasons though is now supported by Coverity.)

No bug is too foolish to check for. Given enough code, developers will write almost anything you can think of. Further, completely foolish errors can be some of the most serious; it's difficult to be extravagantly nonsensical in a harmless way. We've found many errors over the years. One of the absolute best was the following in the X Window System:

```
if(getuid() != 0 && geteuid == 0) {
 ErrorF("only root");
 exit(1);
  }
```

It allowed any local user to get root access[d] and generated enormous press coverage, including a mention on Fox news (the Web site). The checker was written by Scott McPeak as a quick hack to get himself familiar with the system. It made it into the product not because of a perceived need but because there was no reason not to put it in. Fortunately.

False Positives

False positives do matter. In our experience, more than 30% easily cause problems. People ignore the tool. True bugs get lost in the false. A vicious cycle starts where low trust causes complex bugs to be labeled false positives, leading to yet lower trust. We have seen this cycle triggered even for true errors. If people don't understand an error, they label it false. And done once, induction makes the (n+1)th time easier. We initially thought false positives could be eliminated through technology. Because of this dynamic we no longer think so.

We've spent considerable technical effort to achieve low false-positive rates in our static analysis product. We aim for below 20% for "stable" checkers. When forced to choose between more bugs or fewer false positives we typically choose the latter.

Talking about "false positive rate" is simplistic since false positives are not all equal. The initial reports matter inordinately; if the first *N* reports are false positives (*N* = 3?), people tend to utter variants on "This tool sucks." Furthermore, you never want an embarrassing false positive. A stupid false positive implies the tool is stupid. ("It's not even smart enough to figure *that* out?") This technical mistake can cause social problems. An expensive tool needs someone with power within a company or organization to champion it. Such people often have at least one enemy. You don't want to provide ammunition that would embarrass the tool champion internally; a false positive that fits in a punchline is really bad.

Conclusion

While we've focused on some of the less-pleasant experiences in the commercialization of bug-finding products, two positive experiences trump them all. First, selling a static tool has become dramatically easier in recent years. There has been a seismic shift in terms of the average programmer "getting it." When you say you have a static bug-finding tool, the response is no longer "Huh?" or "Lint? Yuck." This shift seems due to static bug finders being in wider use, giving rise to nice networking effects. The person you talk to likely knows someone using such a tool, has a competitor that uses it, or has been in a company that used it.

Moreover, while seemingly vacuous tautologies have had a negative effect on technical development, a nice balancing empirical tautology holds that bug finding is worthwhile for anyone with an effective tool. If you can find code, and the checked system is big enough, and you can compile (enough of) it, then you will always find serious errors. This appears to be a law. We encourage readers to exploit it.

Acknowledgments

We thank Paul Twohey, Cristian Cadar, and especially Philip Guo for their helpful, last-minute proofreading. The experience covered here was the work of many. We thank all who helped build the tool and company to its current state, especially the sales engineers, support engineers, and services engineers who took the product into complex environments and were often the first to bear the brunt of problems. Without them there would be no company to document. We especially thank all the customers who tolerated the tool during its transition from research quality to production quality and the numerous champions whose insightful feedback helped us focus on what mattered. C

References

1. Ball, T. and Rajamani, S.K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth international SPIN Workshop on Model Checking of Software* (Toronto, Ontario, Canada). M. Dwyer, Ed. Springer-Verlag, New York, 2001, 103–122.
2. Bush, W., Pincus, J., and Sielaff, D. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience 30*, 7 (June 2000), 775–802.
3. Coverity static analysis; http://www.coverity.com
4. Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 57–68.
5. Edison Design Group. EDG C compiler front-end; http://www.edg.com
6. Engler, D., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Conference on Operating System Design & Implementation* (San Diego, Oct. 22–25). USENIX Association, Berkeley, CA, 2000, 1–1.
7. Flanagan, C., Leino, K.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 234–245.
8. Foster, J.S., Terauchi, T., and Aiken, A. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 1–12.
9. Hallem, S., Chelf, B., Xie, Y., and Engler, D. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 69–82.
10. Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference* (Berkeley, CA, Jan. 20–24). USENIX Association, Berkeley, CA, 1992, 125–138.
11. Xie, Y. and Aiken, A. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal, Sept. 5–9). ACM Press, New York, 2005, 115–125.

Al Bessey, **Ken Block**, **Ben Chelf**, **Andy Chou**, **Bryan Fulton**, **Seth Hallem**, **Charles Henri-Gros**, **Asya Kamsky**, and **Scott McPeak** are current or former employees of Coverity, Inc., a software company based in San Francisco, CA.; http://www.coverity.com

Dawson Engler (engler@stanford.edu) is an associate professor in the Department of Computer Science and Electrical Engineering at Stanford University, Stanford, CA, and technical advisor to Coverity, Inc., San Francisco, CA.

d The tautological check `geteuid == 0` was intended to be `geteuid() == 0`. In its current form, it compares the address of `geteuid` to 0; given that the function exists, its address is never 0.