The TRUTH About How CodeSonar Compares to PC-lint

Gimpel Software

Version 1, May 2016

As the recognized leader and standard bearer in the static analysis industry, we are flattered to be considered as a benchmark in many product comparisons. While we rarely find it necessary to respond to them, the Grammatech "white paper" comparing the CodeSonar product to PC-lint has the odious distinction of being an exception. Instead of focusing on the perceived merits of the Grammatech product, the white paper in question resorts to false claims and misrepresentations about PC-lint, perhaps as a result of marketplace pressure, which we feel should be balanced with a healthy dose of reality.

About PC-lint

PC-lint is a powerful and reputable C and C++ static analysis tool created by Gimpel Software in 1985 and continually improved during the past 30 years. During this time PC-lint has been an innovative leader in the industry by incorporating pioneering features such as inter-function and inter-module Value Tracking, Strong Type Checking and Dimensional Analysis, and User-Defined Function Semantics. PC-lint is trusted by many tens of thousands of developers, QA analysts, software testers, and forensic analysts and supports dozens of compilers and platforms, sports dozens of secondary features, and is deployed worldwide in virtually every industry including safety critical fields such as medical and automotive.

About Grammatech's white paper

The white paper entitled "How CodeSonar Compares To PC-lint (And Similar Tools)", available from Grammatech's website, purports to provide a comparison between the CodeSonar product and PC-lint (other tools are discussed but the focus is on PC-lint) but in reality is nothing more than a series of self-serving, misleading statements having little or no factual basis. These false claims are then used to attempt to bolster the image of the CodeSonar product at PC-lint's expense.

Refutation Approach

There are several tactics employed by the white paper in an attempt to unfairly disparage PC-lint including:

- Implying PC-lint's technology and feature set has not significantly evolved over the lifetime of the product.
- Intentionally misrepresenting the product goals and feature set of PC-lint.
- Making false claims about PC-lint's capabilities.

We'll break these down into two parts. In the Accusations section, we'll talk about the specific claims made throughout the white paper that, for the most part, are made without any evidence and we'll provide the facts. In the Misrepresentations by Example section, we'll take a look at the examples used in the white paper and provide the actual output of PC-lint which refutes the claims made within those examples. The white paper also contains a handful of legitimate criticisms which we'll explore in the section Legitimate Criticisms.

Accusations

The Grammatech white paper makes several relatively vague and wildly inaccurate, assertions including:

- *Static analysis tools for finding programming problems have been around for decades. The early generation tools, such as those in the lint family are nowadays considered quite primitive. This includes commercial tools such as PC-lint and open-source tools such as Cppcheck. In the past few years, these tools have been superseded by advanced tools such as CodeSonar.*

While we agree the original Unix "lint" tool was quite primitive, trying to make a connection to the primitiveness of that tool based on the similar name of PC-lint is, at best, disingenuous. For over 30 years, Gimpel Software has been an innovative leader in the static analysis realm and PC-lint has been the benefactor of numerous technological advances over the years.

- *CodeSonar's primary purpose is to find serious defects in large code bases. The primary purpose of those early generation tools is much more modest; they are mostly for finding violations of superficial coding standards, and enforcing stronger type checking.*

There is again an attempt to lump PC-lint with these "early generation tools" along with the pretentious claim that PC-lint's goals are "much more modest". PC-lint's primary purpose is to find software defects in both small and large projects including "real" bugs such as buffer overflows, out of bounds access, logic errors, and undefined behavior. PC-lint also aims to serve the needs of our diverse customer base which extend beyond those features provided by much of the competition. This includes support for the various MISRA standards, Strong Type Checking, User-defined semantics, etc. Recognizing that no tool can realistically find all bugs, PC-lint's goals are more lofty. In addition to finding serious defects, PC-lint aims to identify the types of practices leading to bugs in the first place. PC-lint's wide range of features shouldn't be taken to imply a lack of capability when it comes to finding complex bugs.

- *Although the early generation tools claim to find some instances of the serious bugs, in practice they are incapable of finding all but the most obvious.*

Again, while this claim may be true of the original lint, it certainly isn't true of PC-lint and implying this correlation is both disingenuous and misleading.

After providing a series of inter-function examples and claiming PC-lint cannot find any of the bugs (but actually finds almost all the bugs mentioned and some not mentioned by the white paper), the following claim is made:

- *All of the above examples are exceedingly simple. Real code is unfortunately much more complicated; there are many compilation units, lots of levels of abstraction, and very intricate aliasing relationships between variables. If the superficial early-generation tools are incapable of finding fairly obvious defects in tiny examples, they are essentially useless at finding the kinds of serious problems that occur in real life. CodeSonar on the other hand employs a number of sophisticated techniques to model programs, so it is capable of finding real defects.*

This statement starts with the false premise that PC-lint was unable to detect the bugs presented. The white paper then goes on to list some of their distinguishing "techniques" when in reality PC-lint employs the same types of techniques and in some cases has done so for longer than CodeSonar has existed, despite the claim "*None of the primitive tools use any of these techniques.*"

In discussing the user interface, the claim is made:

- *There are many big differences between the user interfaces for these products. The early-generation tools such as PC-lint or Cppcheck were originally designed to be run from the command line, just like a compiler. Consequently they report their results as text output. Although there are integrations with tools that provide higher-level user interface features, these are weaker than a user interface that is completely integrated with the analysis tool. Some examples are given below.*

The first thing to note is the output of PC-lint is extremely customizable and out-of-the box support is provided for textual, HTML, and XML reporting. Regarding graphic interfaces, PC-lint's approach has been to provide the tools necessary to integrate the product with existing tools. We provide configurations for integrating PC-lint with many popular IDEs and there are several third-party companies providing sophisticated integrations with IDEs such as Visual Studio and Eclipse.

The white paper concludes with:

- *The early generation static analysis tools such as PC-lint or Cppcheck use technology that has not changed much in 30 years. As such they are ineffective at finding serious programming defects. Users who continue to use such tools are failing to take advantage of decades of advances in program analysis technology.*

Like much of the white paper, there is absolutely no truth in these three statements. These are complete fabrications supported only by Grammatech's previous false and misleading statements.

Misrepresentations by Example

Note about the examples

The examples provided in the white paper are often incomplete, presumably for demonstration purposes, which does not help when trying to make concrete comparisons or verify the claims made. Most examples referenced have therefore been “cleaned up” so they may be used as complete, stand-alone examples (e.g. on our Online Demo) by placing the snippet inside of a function, declaring types or functions not defined, correcting typos, etc. In all cases, the changes do not affect the semantics or PC-lint’s ability to detect a bug, but simply make it easier for a third-party to reproduce the results described.

Buffer Overrun (static)

```
1   void test_buffer_overrun(int p[]) {
2       p[4] = 1729;
3   }
4
5   void test_driver(void) {
6       int test[4];
7       test_buffer_overrun(test);
8   }
```

The claim made here is:

PC-lint does not detect this because it does not model how information flows across procedure boundaries.

This is simply a false claim. Fifteen years ago PC-lint introduced inter-function value tracking, the functionality the white paper claims is not present. In order to allow the programmer to balance the needs of the project with the hardware resources available, PC-lint utilizes a “multiple-pass” model allowing the user to control the depth in which to search for these kinds of issues. The default depth is 1 whereas a depth of 2 is required to find most inter-function issues. This is discussed throughly in the Reference Manual accompanying PC-lint, and is explained in our website and Online Demo. Running PC-lint with the option `-passes=2` (which is done automatically in the Online Demo) results in the following output:

```
During Specific Walk:
  line 7: test_buffer_overrun([4]) #1
2  Warning 415: Likely access of out-of-bounds pointer (1 beyond end of
    data) by operator '[' [Reference: file ipa2.c: lines 2, 7]
2  Info 831: Reference cited in prior message
7  Info 831: Reference cited in prior message
```

Message 415 warns of the overflow in question along with how far beyond the end of the array the overflow ocurred, as well as the locations leading to this conclusion. The last two messages (831) are optional and are used to provide reference information in a standardized form more easily digestible by IDEs, etc. The following examples will elide the 831 message output (via the `-e831` option) for considerations of space since the same information is included in the warning message.

Buffer Overrun (dynamic)

```
1  typedef unsigned long size_t;
2  void* malloc(size_t);
3  
4  void test_buffer_overrun(int p[]) {
5      p[4] = 1729;
6  }
7  
8  void test_driver(void) {
9      int *p = malloc(4);
10     test_buffer_overrun(p);
11 }
```

The white paper makes the same erroneous claim regarding PC-lint's ability to detect such issues:

PC-lint does not detect this for the same reason it does not detect the overrun of the statically-allocated buffer above.

Running PC-lint with `-passes=2` results in:

```
During Specific Walk:
  line 10: test_buffer_overrun([1]? | 0?) #1
5  Warning 662: Possible creation of out-of-bounds pointer (4 beyond
    end of data) by operator '[' [Reference: file ipa3.c: lines 5, 9, 10]

During Specific Walk:
  line 10: test_buffer_overrun([1]? | 0?) #1
5  Warning 613: Possible use of null pointer 'p' in left argument to
    operator '[' [Reference: file ipa3.c: lines 9, 10]

During Specific Walk:
  line 10: test_buffer_overrun([1]? | 0?) #1
5  Warning 661: Possible access of out-of-bounds pointer (4 beyond end
    of data) by operator '[' [Reference: file ipa3.c: lines 5, 9, 10]
```

PC-lint will diagnose both the creation, and the use, of the out of bounds pointer as well as the possibility the pointer could be null (`malloc` can return null and the example doesn't check for this possibility).

The call portrayal, `test_buffer_overrun([1]?  | 0?)  #1`, given in the verbosity message before the diagnostic provides information about the walk leading to the diagnostic. In this case, the function call being examined is `test_buffer_overrun` which was called with a pointer that either points to an array of one element (`[1]?`, e.g. a single `int`) or is a null pointer (`0?`). The question marks indicate we don't know which is the case since the value was not checked for null. PC-lint doesn't just diagnose the issue, it tells you exactly how it came to the conclusion.

Null Pointer Dereference

```c
#define NULL (void *)0

void test_deref(int *p) {
    *p = 55;
}

void test_driver(void) {
    int *pi1 = NULL;
    test_deref(pi1);
}
```

Again the claim is made that recognizing such an issue is beyond the grasp of PC-lint:

This is probably the simplest possible example of a null-pointer dereference involving two procedures. It is found only by CodeSonar.

And yet again the claim is proven false, again using `-passes=2`:

```
During Specific Walk:
  File ipa4.c line 9: test_deref(0) #1
4  Warning 413: Likely use of null pointer 'p' in argument to operator
    'unary *' [Reference: file ipa4.c: lines 8, 9]
```

PC-lint will report on the null pointer dereference and provides the value tracking history information to back it up.

Memory Leak

```c
typedef unsigned long size_t;
void *malloc(size_t);
void free(void *);

void test_free(int *p, int x) {
    if (p && x < 10)
        free(p);
}
void test_driver(void) {
    int *pi1 = malloc(20);
    test_free(pi1, 20);
}
```

The claim made by the white paper here is:

This example shows code where a buffer is allocated in one procedure, and freed in another, but only if a certain condition is satisfied. It is found only by CodeSonar.

But running PC-lint with `-passes=2` shows this not to be the case:

```
During Specific Walk:
  line 11: test_free([5]? | 0?, 20) #1
8  Warning 429: Custodial pointer 'p' (line 5) has not been freed or
    returned
```

Again, the issue is found by PC-lint, along with detailed information about the call leading to the message.

Legitimate Criticisms

Every tool has strengths and weaknesses and armed with an understanding of a specific tool's weaknesses, it is easy to create contrived examples to highlight those weaknesses. The white paper contains some carefully crafted examples exploiting legitimate, albeit well-known, weaknesses in PC-lint. Most of these examples are due to limitations related to how PC-lint tracks pointer-related information. The Value Tracking system has been enhanced in PC-lint Plus (the next iteration of PC-lint, currently in Beta) and these limitations have been removed. The below examples contain defects which are not diagnosed by PC-lint, instead the output from PC-lint Plus is presented to demonstrate the continual improvements made to the product.

Uninitialized Variables

```c
int foo() {
    int iret;
    int *p = &iret;
    return iret;
}
```

PC-Lint Plus produces:

```
4 warning 530: likely using an uninitialized value
    return iret;
           ^
2  supplemental 891: allocated here
    int iret;
        ^
```

Use After Free

```c
typedef unsigned long size_t;
void *malloc(size_t);
void free(void *);

void foo() {
    char *p = (char *)malloc(10);
7      char *q = p;
8      if (p) {
9          p[0] = 'X';
10         free(p);
11         q[0] = 'Y';
12     }
13 }
```

PC-lint Plus produces:

```
11  warning 449: memory was likely previously deallocated
        q[0] = 'Y';
             ^
10  supplemental 891: deallocated here
        free(p);
        ^
6  supplemental 891: allocated here
    char *p = (char *)malloc(10);
                      ^
```

Double Free

```
1  typedef unsigned long size_t;
2  void *malloc(size_t);
3  void free(void *);
4
5  void test_double_free(int *p) {
6      if (p)
7          free(p);
8  }
9
10 void test_driver(void) {
11     int *pi1 = (int *)malloc(sizeof(int));
12     if (pi1)
13         test_double_free(pi1);
14     if (pi1)
15         free(pi1);
16 }
```

PC-lint Plus produces:

```
15  warning 2432: memory was potentially deallocated
        free(pi1);
        ^
15  supplemental 894: during specific walk free([4]@0/1)
        free(pi1);
        ^
7  supplemental 891: deallocated here
        free(p);
        ^
11  supplemental 891: allocated here
    int *pi1 = (int *)malloc(sizeof(int));
                      ^
```

Buffer Overruns

```c
void foo() {
    char buffer[10];
    char *pc;
    pc = buffer;
    for (int i = 0; i <= 10; i++)
        *pc++ = 'X';
}
```

This example contains a buffer overrun in the `for` loop. The models employed by PC-lint and PC-lint Plus to track value state information do not (yet) take into account the relationship between `i` and `pc` in this example. This is a legitimate limitation of PC-lint's current abilities. While we could easily provide a similarly simple example exploiting the weaknesses of CodeSonar, doing so would not accomplish anything. As mentioned above, we recognize every tool has a unique set of strengths and weaknesses and all tools will be able to catch bugs that others cannot. Instead of highlighting weaknesses in other tools, we are content to focus on the strengths of PC-lint and work to continually improve the product to meet our customers' needs.

Summary

In this table we summarize the key misleading statements made by Grammatech about PC-lint and we provide the corrected information.

Claim	Reality
PC-lint is a primitive tool in the family of the original Unix `lint`.	PC-lint is a state-of-the-art static analysis tool independently developed and continually improved over the last 30 years with award-winning innovations including inter-function and inter-module Value Tracking.
PC-lint fails to detect all but the most obvious bugs.	PC-lint is capable of finding complex bugs using a wide range of leading edge technologies, including bugs in the examples in the white paper.
PC-lint isn't focused on finding serious issues in large projects.	PC-lint is successfully used with projects of all sizes, from hundreds of lines of code, to millions.
PC-lint is a text-only tool not suitable for use with continuous integration tools.	PC-lint can provide reporting in virtually any format including plain text, HTML, and XML and is used by customers in many ways, including integration with CI tools like Hudson and Jenkins.
PC-lint doesn't provide history information when diagnosing complex issues.	PC-lint provides a significant amount of information when warranted, including the series of calls and values leading to a particular determination.
PC-lint is designed only for use by developers.	PC-lint is designed for and used by software developers, QA analysts, software testers, and forensic experts.
PC-lint only understands issues within a single file.	PC-lint has been able to provide whole-program, inter-module analysis since its inception. This was one of the original differentiators of PC-lint.
PC-lint is not able to catch every possible software defect.	While no static analysis tool can claim to catch all defects, PC-lint has a track record as a reliable tool capable of catching many complex, real-world issues and is constantly being improved.

Conclusion

While Gimpel Software welcomes honest feedback and criticism from customers and competitors alike, the false and misleading claims made in the referenced white paper are neither honest nor constructive and do not serve the best interests of programmers or the static analysis community. Relying on false and disparaging claims about competing products is a strategy which reveals more about Grammatech than its competitors.

From the white paper, it appears the main points of notable difference between PC-lint and CodeSonar are:

- CodeSonar has a tightly integrated GUI while PC-lint's focus is on providing extensible mechanisms which can be used to integrate PC-lint with existing tools such as Visual Studio, Eclipse, etc.
- CodeSonar has a more narrow focus which results in a smaller feature set than PC-lint.
- CodeSonar provides several features PC-lint does not yet support. Specifically mentioned are Metrics and Taint Analysis, both features being developed for a future version of PC-lint Plus. Of course, PC-lint contains a large number of features not supported by CodeSonar.