MACHINE LEARNING FOR FINDING PROGRAMMING DEFECTS AND ANOMALIES

TRUSTED LEADERS OF SOFTWARE ASSURANCE AND ADVANCED CYBER-SECURITY SOLUTIONS

WWW.GRAMMATECH.COM

INTRODUCTION

Static analysis tools are useful for finding serious programming defects and security vulnerabilities in source and binary code. Most static analysis checkers work by searching the code for known patterns or conditions that cause the program to fail, or that indicate violations of coding standards. The set of defects that such tools can find is limited to problems anticipated by the tool-designer.

Newer, advanced tools using machine learning techniques can automatically determine new properties to check by deducing normal usage, and then looking for parts of the code that deviate from that practice in significant ways, on the assumption that such abnormal code is incorrect. This approach has previously been limited to the scope of the body of code under analysis, but the ever-increasing volume of open source software, combined with advances in machine learning, means that it is now possible to deduce normal usage by mining very large software collections. This technique is particularly useful for finding anomalies in API usage, especially for popular operating system interfaces or open source libraries. This paper describes how the technique works and shows how it was able to find several previously unknown bugs in high-profile software systems with high precision (i.e., few false positives).

ADVANCED STATIC ANALYSIS

Roughly speaking, advanced static-analysis tools work as follows: First they create a model of the entire program which they do by reading and parsing each input file. The model consists of representations such as abstract-syntax trees for each compilation unit, control-flow graphs for each subprogram, symbol-tables, and the call graph. Checkers that find defects are implemented in terms of various kinds of queries on those representations. Superficial bugs can be found by doing pattern matching on the abstract syntax tree or the symbol tables. The really serious bugs are those that cause the program to fail, or that lead to security vulnerabilities, such as null pointer dereferences, buffer overruns, etc., and these require sophisticated queries to find. Those queries can be thought of as abstract simulations — the analyzer simulates the execution of the program, but instead of using concrete values, it uses equations that model the abstract state of the program, and when anomalies are encountered, warnings are generated.

The kinds of defects fall into three main categories:

1. Bugs that violate the fundamental rules of the language, thereby causing the program's behavior to be undefined. These includes memory errors such as null pointer dereferences and buffer overruns, concurrency errors such as data races, and bugs such as use of uninitialized memory.
2. Defects that arise because the program breaks the rules of using a standard API. For example, the C library does not specify what happens when the same file descriptor is closed twice; this makes no sense to do deliberately so is probably a bug. Mismanagement of finite resources such as dynamically-allocated memory also fall into this category.

3. Inconsistencies or contradictions in the code. These may not cause the program to crash, but likely indicate that the programmer misunderstood an important property of the code. For example, a condition that is either always true or always false is unlikely to be intentional because it leads to dead code. Most violations of coding standards such as MISRA C fall into this category.

Many tools give users the ability to define custom checkers, such as for properties that arise from their specific problem domain, or to find violations of proprietary coding standards.

This paper is mostly about finding defects in the second category — API misuse. The next section describes how static analysis tools find such bugs and goes on to describe how machine learning techniques can be employed to learn what “normal” use is, and then demonstrates how this can be used by static analysis tool to find real and significant defects due to abnormalities in usage.

STATIC ANALYSIS FOR API MISUSE

All but the most trivial programs use external libraries, either commercial, open source, or in-house libraries. These libraries are typically called through Application Programming Interfaces, or APIs for these. These APIs define the interfaces to libraries of code. These can include the C library, operating system libraries, and interfaces to algorithm collections, networking stacks, cryptographic services, or windowing systems. All of these APIs have rules about what is valid usage. Advanced static analysis tools model the run-time behavior of programs, so they must have knowledge of these APIs if they are to find bugs caused by their misuse. If the library is available as source code, it can be analyzed alongside the code that uses it, increasing the scope of the analysis to include the source from the libraries. If the source is not available then binary code static analysis tools can be used. However, if neither of these are feasible, knowledge of the API is typically acquired in several ways:

- Calls to functions in the API may be treated as if they were mostly no-ops. This means the tool is essentially blind to what the functions do, and this may cause false positives (warnings that are not real bugs) or false negatives (bugs that go undetected.)
- Each function in the API can be modeled to simulate its essential semantics. Tool vendors do this in advance for widely used libraries, and end users can do it too, but it can be very tricky to get right and is labor intensive.
- Observe how the function is used from a large set of examples, and infer what its correct usage is. This is essentially a machine learning activity.

Most current static analysis tools use a blend of the first two of these options. For example, they have some understanding of commonly used APIs such as POSIX, the Windows API and some real-time operating systems’ APIs. However, there are so many APIs that it is impractical for tool writers to cover them all, and so the tool may be unable to find defects associated with their code.

The last of the options—observing an API in use—is best illustrated by an example. Suppose there is an API function that returns a value. If that value is checked by the caller 99% of the time, then the remaining 1% of calls that do not check the value, simply by virtue of their scarcity, may be worth reviewing in case they are oversights or mistakes. Experience has shown that those anomalous instances do correlate reasonably well with real defects. Because it is very common for return values to indicate error conditions, failure to confirm that the function succeeded can mean a serious bug is lurking in the code. A good example is shown in Figure 1.

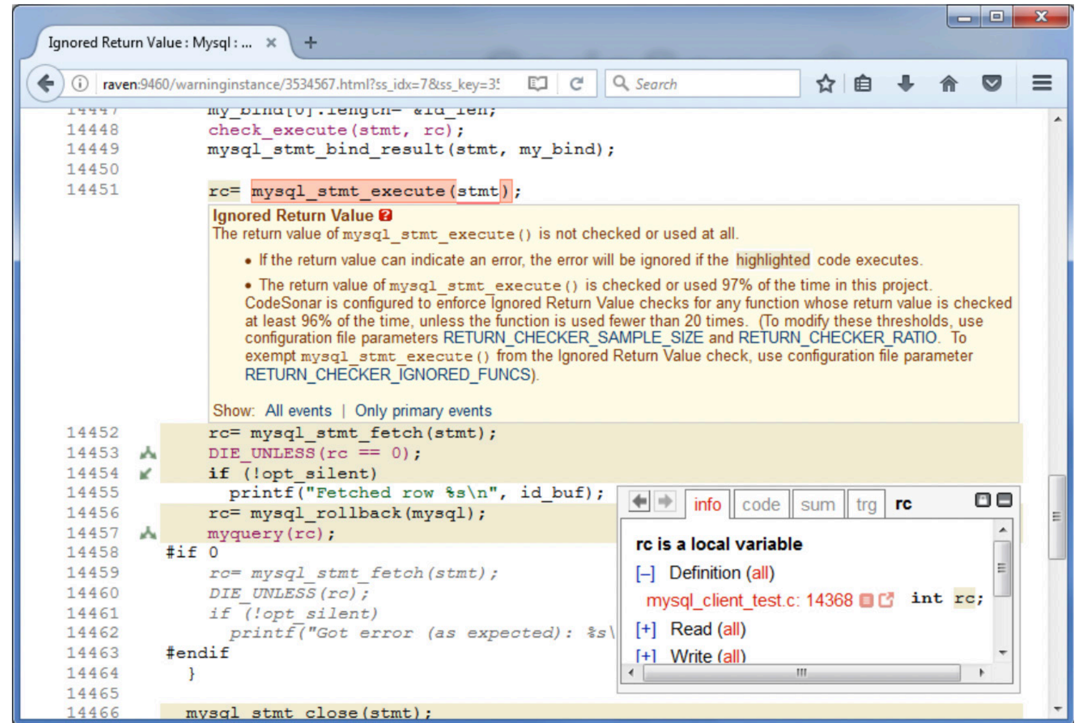

Figure 1. A warning report from CodeSonar showing an ignored return value. The accompanying text explains why the analysis has concluded that the given call is an anomaly. Note that line 14452 overwrites the value of rc before it can be checked, and that line 14453 checks that value, all of which strengthens evidence that ignoring the value may be hazardous.

A weakness of this approach is that the analysis needs enough samples of usage to be able to determine what is statistically significant to report. If the program had only five calls to a function, and in only one of them was the return value ignored, then it is unreasonable to conclude that single instance is a real anomaly.

This weakness is avoided if more code is analyzed. If analyses of hundreds of other programs that use that function from an API yields hundreds of instances where the return value is checked, and very few where it is not checked, then it is more reasonable to say that the missing checks are likely to be wrong, even if there are only a small number of usages in the target program.

The next section describes how this technique of mining large quantities of code to find patterns of normal usage is generalized. The subsequent section then describes how this can be used to statically identify anomalies.

LEARNING NORMAL USAGE

A key principle of this approach is to exploit the observation that most code is correct most of the time and that code that is more mature is more correct than code that has just been written. Consequently, if large-scale machine learning can be used to observe how APIs are normally used in a large corpus of real programs, then it can be possible to find bugs by locating places where the use is anomalous.

The example in the previous section was illustrated the detection of ignoring the return value of a function. There are other properties of functions that are useful to know about. Many have to do with the value or type of parameters that are passed in, or the value that gets returned. For example:

- For scalar-typed values, what are the valid range of values? For some functions it is normal for only constants to be used for some of the parameters.
- For pointer-typed values, are they allowed to be NULL? Are they always dereferenced, or only sometimes? Do they point to dynamically-allocated memory? Might they free that memory? Are they expected to represent null-terminated strings?
- Are there correlations between properties of values? For example, a function may return NULL if and only if its first parameter is NULL.

To implement this approach, GrammaTech built a framework capable of iterating over a large corpus of programs. The architecture is as shown in Figure 2.

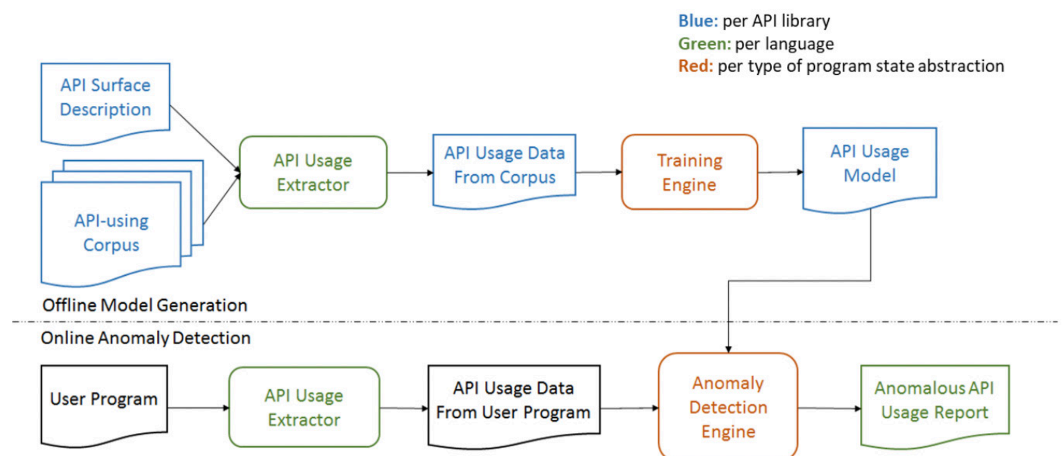

Figure 2. A block diagram showing the architecture of the system created for mining information about API usage from a large corpus of sample programs.

A key component in this framework is the API Usage extractor. CodeSonar was used to do this because it can parse programs in a number of different dialects of C and C++, and it has a convenient API for extracting semantic information about the call site.

GrammaTech ran CodeSonar on the Fedora Linux SRPM (source) repository as our initial C/C++ corpus for feature extraction. The size of this corpus, after performing simple deduplication, was 14,382 projects, containing over 1.4 billion lines of code. After additional, more sophisticated deduplication that required static analysis, our final corpus contains approximately 7,000 projects. This covered over 1500 API functions, belonging to widely-used APIs such as the GNU C Library, OpenSSL, Qt, Python extension API, GLib GNOME Library, GTK, libPNG, XLib, libXML, etc.

STATIC API ANOMALY DETECTION

Having mined the information about API usage from the corpus, and having used machine learning to acquire knowledge about common usage, the results were fed into CodeSonar to find new defects. This part of the system corresponds to the bottom half of Figure 2 above.

We give an example below in Figure 3.

```
1483            }
1484            param_str = qof_util_param_to_string (ent, inst->param);
1485            if (param_str)
1486                    g_strescape (param_str, NULL);
1487            sql_str = g_strconcat ("UPDATE ", ent->e_type, " SET ",
1488                    inst->param->param_name, " = '", param_str,
1489                    "' WHERE ", QOF_TYPE_GUID, "='", gstr, "';", NULL);
1490            LEAVE ("sql_str=%s", sql_str);
1491            return sql_str;
1492    }
```

Figure 3. A fragment of a report from CodeSonar showing a newly-detected defect involving the use of an API function named g_strescape. In this case the programmer seems to have misunderstood the interface to the function.

This code comes from a Linux application package called QOF (Query Object Framework.) The programmer ignores the return value of `g_strescape` and uses the unescaped `param_str` in subsequent code that constructs a SQL database update statement. This creates an opportunity for an SQL injection attack, with potential for serious security consequences.

In addition to this defect, GrammaTech identified other, previously undiscovered, defects in many different projects. The precision of this analysis is good so while there are some inevitable false positives, they are worth tolerating in order to find the real defects.

This machine learning technique is sufficiently general that it can be applied in many other situations. Although the examples described above targeted programs that call widely-used open-source APIs, there is no reason why it could not learn and check API usage from a corpus of proprietary third party, or in-house, closed-source programs, as long as the corpus is large enough and there are sufficient samples of calls to those functions.

CONCLUSION

Applying machine learning techniques to a large corpus of source code shows that it is feasible to use as way to find anomalies in API usage. Experience shows that a significant proportion of such anomalies correspond to real defects, some undiscovered until this work was done. The next release of CodeSonar incorporates this technique, which increases the range of bugs that it can find statically.

GrammaTech, Inc. is a leading developer of software-assurance tools and advanced cyber-security solutions. GrammaTech helps organizations develop and release high quality software, free of harmful defects that cause system failures, enable data breaches, and increase corporate liabilities in today's connected world. GrammaTech's CodeSonar is used by embedded developers worldwide.

CodeSonar and CodeSurfer are registered trademarks of GrammaTech, Inc.
© 2019 GrammaTech, Inc. All rights reserved.