Proceedings of Static Analysis Summit II

Paul E. Black (workshop chair)
Elizabeth Fong (editor)
Information Technology Laboratory
U.S. National Institute of Standards and Technology (NIST)
Gaithersburg, MD 20899

Static Analysis Summit II (http://samate.nist.gov/SASII) was held 8 and 9 November 2007 in Fairfax, Virginia,and was co-located with SIGAda 2007. A total of 61 people registered, coming from government, universities, tool vendors and service providers, and research companies. The workshop had a keynote address by Professor William Pugh, paper presentations, discussion sessions, a panel on "Obfuscation Versus Analysis – Who Will Win?", and a new technology demonstration fair. The workshop is one of a series by NIST's Software Assurance Measurement and Tool Evaluation (SAMATE) project, which is partially funded by DHS to help identify and enhance software security assurance tools.

The Call for Papers pointed out that "Black-box" testing cannot realistically find maliciously implanted Trojan horses or subtle errors with many preconditions. For maximum assurance, static analysis must be applied to all levels of software artifacts, from models to source code to binaries. Static analyzers are quite capable and are developing quickly. Yet, developers, auditors, and examiners could use far more capabilities. The goal of this summit is to convene researchers, developers, and government and industrial users to define obstacles to such urgently-needed capabilities and try to identify feasible approaches to overcome them, either engineering or research.

The Call for Papers solicited contributions describing basic research, applications, experience, or proposals relevant to static analysis tools, techniques, and their evaluation. These proceedings include the agenda, some discussion notes, and reviewed papers.

We are especially grateful to Prof. William Pugh for the enlightening keynote address. I thank those who worked to organize this workshop, particularly Wendy Havens, who handled much of the correspondence. We appreciate the program committee for their efforts reviewing the papers. We are grateful to NIST, especially the Software Diagnostics and Conformance Testing Division, which is in the Information Technology Laboratory, for providing the organizers' time. On behalf of the program committee and the SAMATE team, thanks to everyone for taking their time and resources to join us.

Dr. Paul E. Black
February 2008

Static Analysis Summit II Agenda

Thursday, 8 November, 2007

12:45 :	Static Analysis for Improving Secure Software Development at Motorola - R Krishnan (Motorola), Margaret Nadworny (Motorola), and Nishil Bharill (Motorola)
1:10 :	Discussion: most urgently-needed capabilities in static analysis
1:40 :	Evaluation of Static Source Code Analyzers for Real-Time Embedded Software Development - Redge Bartholomew (Rockwell Collins)
2:05 :	Discussion: greatest obstacles in static analysis
2:50 :	Common Weakness Enumeration (CWE) Status Update – Robert Martin (MITRE) and Sean Barnum (Cigital)
3:15 :	Discussion: possible approaches to overcome obstacles
3:45 :	Panel: Obfuscation vs. Analysis - Who Will Win? – David J. Chaboya (AFRL) and Stacy Prowell (CERT)
4:30 :	New Technology Demonstration Fair FindBugs, FX , Static Analysis of x86 executables

Friday, 9 November, 2007

8:30 AM:	Discussion: Static Analysis at Other Levels
9:00 :	Keynote: Judging the Value of Static Analysis - Bill Pugh (UMD) The slides for the keynote address are on-line at http://www.cs.umd.edu/~pugh/JudgingStaticAnalysis.pdf
10:15 :	A Practical Approach to Formal Software Verification by Static Analysis - Arnaud Venet (Kestrel Technology)
10:40 :	Discussion: inter-tool information sharing
11:10 :	Logical Foundation for Static Analysis: Application to Binary Static Analysis for Security - Hassen Saidi (SRI)
11:35 :	Wrap up discussion: needs, obstacles, and approaches

Discussion and Panel Notes

To catalyze discussion, we presented six questions or topics. The discussions were not meant to reach a consensus or express a majority, and seldom did they. Workshop participants presented ideas, questions, recommendations, cautions, and everything in between. Although we try here to note what was said, it is by no means a complete record of what was discussed (none of us wrote fast enough). In some cases we combined similar comments across sessions. We hope these notes convey some feel for the discussions and lead to improvement.

1:10 PM Most Urgently-needed capabilities in Static Analysis

Facilitator: Vadim Okun, NIST

- Be able to analyze
  - Concurrent/race conditions
  - Timing
  - Runtime dispatch, deep hierarchies, highly polymorphic systems
  - Function pointers
  - Integer overflow
  - Numeric computations accurately (comment was: Inaccurate numeric analysis)
  - Inline assembly with multi-languages across multi-boundaries
- General capabilities needed/capabilities missing
  - Lack of reasoning for reporting
  - Reduce false positives
  - Report the probability that a weakness is exploitable.
  - Tool reports what its coverage, that is, what it looks for
  - Scalability (e.g., 100 million lines)
  - Whole application analysis
  - A standard way to express environment in a standard way so one can state what is known, so the next tool doesn’t have to do the same thing.
- Need guarantee that if we run this tool, we will not have these exploits.

2:05 PM Greatest Obstacles in static Analysis

Facilitator: Paul E. Black, NIST

Need scientific surveys or studies that show return on investment of tools.

Its likely there is at least one tool for each of the things on the wish list. We need a toolbox.

Tools do not know what the requirements are, what the program is supposed to do. To go beyond buffer overflow, which is (almost) always a violation, tools need a specification, like IFIL. Java has JML, and there is Splint for C, but we can’t get people to use annotations! We need an easier way to write annotations.

Programmers need skill sets for good code creation; they should be reinforced by tools.

The European safety community has used static analysis for years. They made their case.

3:15 PM Possible Approaches to Overcome Obstacles

Facilitator: Redge Bartholomew, Rockwell Collins

Vendor should provide information about what exactly the tools find.

Many people are skeptical about using test sets, particularly fixed sets, to evaluate tools. Tools get tailored to test suites.

You can't include a tool in certification efforts until it is very well qualified. Tests are necessary, but not sufficient.

There was a discussion about funding better software, both research and development of techniques and paying well for good quality software. Rod Chapman said, you cannot polish junk. (That is, software must be built well from the beginning. No amount of tools or techniques can "repair" poor software.) He also said, if all software is junk, we might as well buy cheap junk. (That is, if consumers can't judge the quality of software, it is logical in today's world to assume the worst of software. Therefore people won't pay much for software tools or programs.)

3:45 PM Panel: Obfuscation vs. Analysis – Who Will Win?

Malware writers use obfuscation to disguise their programs and hide their exploits. Good guys need powerful analysis to crack malware quickly. Good guys also use obfuscation to protect intellectual property, and in military applications, hinder enemies from figuring out weapon systems (remember the Death Star?). They don't want bad guys to crack their techniques. This panel was set up to explore who will win and why.

The panelists gave very good presentations, but instead of entertaining controversy, they agreed that analysts ultimately win.

8:30 AM Static Analysis at Other Levels

Facilitator: Michael Kass, NIST

Here are other static analysis applications or targets, in addition to the "default" source code analysis for bugs:

- Requirement analysis (lots of resistance from implementers because it is yet another language to learn, but probably will happen)
- Architectural design review
- Compiler/Decompilers
- Code metrics generation, e.g., measuring code
- Program understanding

Reverse engineering (e.g., byte code to UML design)
Re-engineering (e.g., re-factoring)
Program/property verifier
Binary analysis (they are as good as source analyzers if you have the symbol table)

The audience suggested more static analysis tools:
Source to source transformers
Same language translator (e.g., debugger, dissembles, emulators)
Threat modeling tools
Impact analysis/slicers
Model checker (it is not useful unless there is some manual checking)
Combining static and dynamic analysis (e.g. static analysis plus testing, static analysis and program verification)

10:40 AM Inter-tool Information Sharing

Facilitator: Paul E. Black, NIST

The most important requirement for inter-operation of tools is to have common reporting format. Many companies have more than one type of tool, and to facilitate integration among these different tools in a user-friendly environment, it is useful to have one tool's output become another tool's input.

To promote progress, here are some use cases for information sharing:
Generic format to explain "reasoning" of a bug report
SA tool -> infeasible paths -> testing
SA tool -> no alias in blocks, etc. -> compiler optimization

11:35 AM Wrap Up discussion: Needs, Obstacles, and Approaches

Facilitator: Paul E. Black, NIST

The biggest need is for people to agree on what content to share and common report formats. One needs to get information from runs into static analysis as the basis of hints, hypotheses, or invariants. We need to identify use cases for information sharing. The biggest challenge today is for tool to explain (to another tool or to human) the following:
The complicated path and the "reasoning" or evidence of a bug report.
The information provided to an assurance case (e.g., guaranteed no SQL-injection)
What areas (either code blocks or types of problems) are NOT analyzed.

Such work needs to address the different needs of auditors, assessors and developers.

The recommendation is for NIST to conduct a tool exposition. Tool vendors should sign-up to run their tools with NIST's selected test source programs.

A burning issue is an effective way to get feedback from users about tools.

Static Analysis Tools for Security Checking in Code at Motorola

R Krishnan, Margaret Nadworny, Nishil Bharill
Motorola Software Group
<krishnanr@motorola.com; Margaret.Nadworny@motorola.com; Nishil.Bharill@motorola.com>

Abstract

As part of an overall initiative to improve the security aspects in the software used in Motorola's products, training and secure coding standards were developed. The goal is to decrease the number of security vulnerabilities introduced during the coding phase of the software development process. This paper describes the creation of the secure coding standards and the efforts to automate as many of the standards as possible.

Originally, the efforts focused on the *Inforce* tool from Klocwork, as many Motorola business units already used the tool for quality but without the security flags activated. This paper describes the efforts to evaluate, extend, and create the coverage for the secure coding standards with Klocwork. More recently, an opportunity arose which allowed a team to evaluate other static analysis tools as well. This paper also describes the findings from that evaluation.

Keywords: static analysis, security, Klocwork

Introduction

Security is one of the key product quality attributes for products and solutions in telecommunications. A denial of service attack on a telecom network could mean a huge loss in revenue to the operator. With mobile phones used increasingly for a wide range of services beyond telephony from messaging to online shopping, security has become an important aspect of the software on these devices. Factors such as the increased connectivity of devices and the use of open source software increase the security risks. As a result, Motorola has increased the priority and attention to the security related aspects of its products.

Motorola Software Group is a software development organization existing in the Corporate Technology Office providing software resources and services to all of the Motorola Businesses.

The approach within Motorola Software is to build security into the products throughout the development lifecycle. Changes in software development are institutionalized when they become part of the process, with appropriate tool support and with the engineering community trained on the required tools and process changes. This approach is depicted in Figure-1. Specifically, to instill a security focus in the coding phase, the coding standards are enhanced with security rules, training is required on the basic concepts relating to secure programming, and a static analysis tool is used to automate the identification of any violation of the security rules.

Security Focus in the Coding Phase

The coding phase is recognized as a key phase, where vulnerabilities are introduced by the developers into the code which put the system at risk from attack. The vulnerabilities targeted include buffer overflow and format string vulnerabilities. This area was viewed as requiring relative low effort in terms of changing processes but high impact in improving the security of the products. As a result, the coding phase was the first area of security change within the organization.

Recognized security experts from FSC, now Assurent, a subsidiary of TELUS, were engaged to assist with the development of Secure Programming Training to educate the engineers on the need, importance, and details of Secure Programming. In addition, the Assurent staff assisted in the enhancement of the coding standards for C, C++, and Java with security rules. Previously, the coding standards for quality focused on the readability and maintenance aspects of the code. The security rules introduced significant content, addressing what was and what was not recommended from the security perspectives. The content is segregated into rules which are mandatory and guidelines which are recommended and are optional.

For the C coding standards, twenty-three rules and twenty-one guidelines were introduced and adopted. The rules include the following aspects:

- Buffer Overflows
- Memory allocation and deallocation
- Handling of resources such as filenames and directories
- Use of library functions
- Overflows in computation
- Avoiding format string vulnerabilities
- Input validation

- Handling of sensitive data
- and others.

For the C++ coding standards, thirty two rules and three guidelines were introduced and adopted. The rules cover the following aspects:

- Memory allocation
- Avoiding C-style strings
- Initialization
- Pointer casting
- Use of vectors instead of arrays
- Orthogonal security requirements
- Exceptions
- Use of STL (Software Template Library)
- and others.

For the Java coding standards, sixteen rules and three guidelines were incorporated. The rules include the following aspects:

- Use of secure class loaders
- Object Initialization
- Securing of packages, classes, methods, variables
- Handling of sensitive data
- Random number generation
- Comparison of classes
- and others.

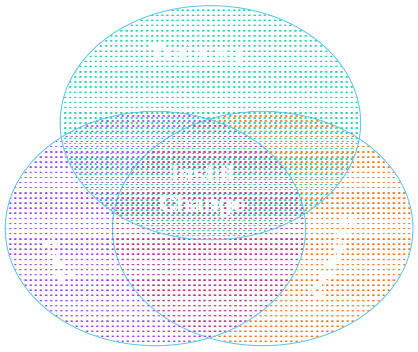


Figure-1: Instill Change in the Implementation Phase

Static analysis tools help in automatically detecting violations of the security rules.

Originally, the efforts focused on the *Inforce* tool from Klocwork as many Motorola Business Units already used the tool for quality but without activated security flags. More information about Klocwork is available on their web site at: www.klocwork.com. No other tools were seriously considered initially. Supporting two different static analysis tools, one for quality and another for security, was not practical for three reasons: licensing costs, productivity inefficiencies and vendor management.

Supporting the Security Rules in Klocwork

The following process was used to collaborate with Klocwork to support the Motorola Coding Standards. The security rules in the coding standards were analyzed and the opportunities for automatic detection of violations to the coding standard were identified. Some rules, by the nature of their content, cannot be verified through static analysis. An example of such a rule is: *"Resource paths shall be resolved before performing access validation"*. This particular rule must be verified through usual inspection practices. This overall analysis of which rules could be automated was a collaborative effort with the Klocwork technical team.

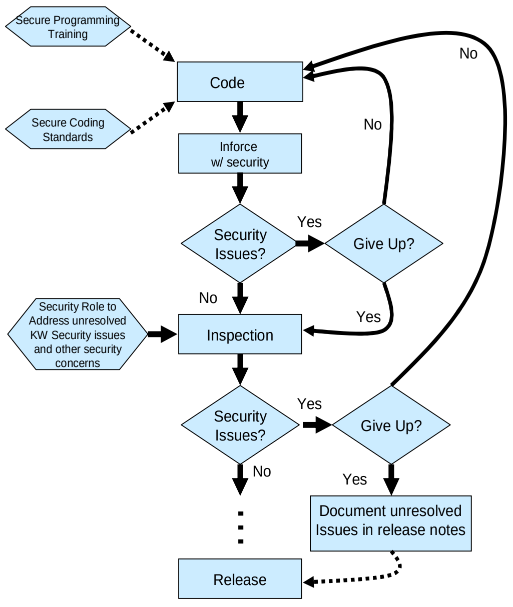

Secure Programming Training
Secure Coding Standards
Code
Inforce w/ security
Security Issues?
Yes
Give Up?
No
No
Security Role to Address unresolved KW Security issues and other security concerns
No
Inspection
Yes
Security Issues?
Yes
Give Up?
No
Yes
Document unresolved Issues in release notes
Release

Figure-2: Use of Klocwork *Inforce* with Security Flags

Test cases were created for each of the rules verifiable through static analysis. Negative testcases create instances of violation of each security rule. The tool is intended to catch and flag these negative errors. These negative test cases check for false negatives. Positive testcases are instances that are in conformance to the rule. The tool is not expected to flag errors on these testcases. These positive testcases check for false positives.

The existing checkers available in Klocwork were analyzed. Gaps between the available checkers and the verifiable security rules were identified. Klocwork has a feature where additional checkers can be created without waiting for the next release of the tool. These extensibility checkers were used to address the identified gaps.

The new extensibility checkers identified were developed and delivered by Klocwork, in a phased manner for C and Java. The success criteria for these extensibility checkers were to detect the violations in the negative testcases and pass the positive testcases. The C extensibility checkers have not been incorporated into Klocwork's general releases due to legal intellectual property related issues, which is also why they are not described in further detail. Activities have been initiated to revisit this situation. These extensibility checkers were delivered to Motorola and confirmed. The situation for Java was handled similarly. The checkers were written by Klocwork but were integrated into Release 7.5, as Klocwork determined that the security rules could be identified from published material. For the C++ security rules, Motorola software group engineers in Bangalore were trained by Klocwork to write extensibility checkers. The Bangalore team developed the checkers in-house.

Table-1 shows the progress made in the Klocwork *Inforce* tool with this activity. The first column provides the programming language. The second column provides the total number of security rules including subrules for the corresponding programming language. The third column provides the total number of security rules and subrules which could be automated. The fourth column indicates the number of rules successfully supported in earlier versions of Klocwork, specifically version 6.1 for C and C++ and 7.1XG for Java. This represents the initial results from this activity. The fifth column provides the number of rules successfully supported since Klocwork 7.5. For C, Klocwork 6.1 supported eight rules in Klocwork which was extended to support twenty-two rules in Klocwork 7.5. For C++, Klocwork 6.1 supported two rules which were extended to support nineteen rules in Klocwork 7.5. For Java, Klocwork 7.1XG supported two rules in Klocwork which was extended thirteen rules in Klocwork 7.5. The improvement has been impressive but is by no means complete.

Klocwork Benchmarking Activity

Buffer overflow is one of the most dangerous coding vulnerabilities in software that continues to be exploited. It has obtained the attention of researchers as well, including the Software Assurance Metrics and Tools Evaluation (SAMATE) [4] project, sponsored by the U.S. Department of Homeland Security (DHS), National Cybersecurity Division and NIST.

The required scripts were developed to utilize the test cases/code snippets offered by the MIT Test suite [1] and were passed through Klocwork with all the errors enabled. Overall, five defects were identified in the Klocwork tool itself. Change requests have been submitted to Klocwork and the errors will be addressed in the upcoming 8.0 release of Klocwork. The defects identified include:

- Violation on access to shared memory
- Function call used as an array index with return value exceeding array bounds
- Array element value used as index in accessing another array exceeding array bounds
- Use of function call in strncpy for the value of n, exceeding array bounds
- Accessing beyond bounds, after assigning the array start address to a pointer variable.

SAMATE[4] provides a set of testcases for different languages like C, C++, Java, PHP, etc. A study was performed to understand the coverage of the security rules in the Motorola coding standards in this reference data set. There are 1677 testcases for C and 88 testcases for C++ in the SAMATE reference dataset. These testcases cover aspects such as memory leaks, double free memory errors, input validation, buffer overflow of both stack and heap, null dereference, race condition, variable initialization, command injection, cross-site scripting, format string vulnerabilities and so on. There was considerable overlap between the Motorola test suite and the SAMATE test suite. Thirteen of the security rules in the Motorola Software C coding standard are covered in the SAMATE test set. Four of the security rules in the Motorola Software C++ coding standard are covered in the SAMATE test set.

There are thirty-three testcases for Java in the SAMATE reference dataset. These testcases cover aspects such as tainted input, arbitrary file access, tainted output, cross-site scripting, memory resource leaks, and return of private data address from public methods. There are no overlaps with the security rules in the Motorola Software Java coding standard. This is summarized in Table-2.

One of the major shortcomings identified with Klocwork was its inability to identify the use of uninitialized array variables. The Klocwork team has analyzed and identified particular aspects of this general problem:

Language	Number of Security Rules	Number of Automated Rules	Support in Klocwork 6.1	Support in Klocwork 7.5
C	39	25	8	22
C++	34	25	2	19
Java	16	5	9	13

Table-1: Security Rules support in Klocwork

- Uninitialized use of array elements of simple variable type
- Uninitialized use of array elements of complex variable type such as arrays of structures or pointers
- Uninitialized use of global arrays
- Partial initialization determination: being able to identify that some elements are initialized and some are not
- Interprocedural initialization with initialization occurring in a different function.

Factors like complex data types, global array variables, partial initialization, and the need for interprocedural analysis make detection of uninitialized use of array elements technically difficult for static analysis tools. Klocwork promises to provide a phased solution over the next couple of releases.

Language	Number of SAMATE Tests	Number of Motorola Rules Covered
C	1677	13
C++	88	4
Java	33	0

Table-2: SAMATE Testcases and Motorola Coding Standard Rules.

The inability to identify uninitialized array elements was the root cause for issues identified in field testing of some of Motorola products. The initial response from Klocwork, after reporting this problem, triggered the assessment of other popular security static analysis tools with Klocwork. As the intention of this paper is to create improvement in all security static analysis tools, the names of the other tools will be referenced here as Tool X, Tool Y, and Tool Z. Table-3 shows the comparison of these other tools with the Motorola developed testcases as the basis for comparison. The first column represents the programming language evaluated. The second column provides the total number of security rules including subrules. The third column provides the Klocwork results for release 7.5.

Language	Number of Security Rules	Klocwork 7.5	Tool X	Tool Y	Tool Z
C	39	22	7	7	5
C++	34	19	0	1	1
Java	16	13	2	1	0

Table-3: Support for Motorola Security Rules in Static Analysis Tools.

The last three columns show the results from three well known security static analysis tools. Detailed results for this benchmarking activity can be found in Appendix A for C, Appendix B for Java, and Appendix C for C++. Because none of the positive test cases detected any false positives in this activity, this paper does not elaborate further.

Observations:

- Klocwork is significantly better supporting the security rules in the Motorola Coding standards due to the collaboration.
- Our partnership with Klocwork has been a major factor in the support to these security rules in their tool suite.
- Support for detecting uninitialized use of array elements is weak in the major static analysis tools for security. Tool X and Klocwork could handle detection of uninitialized use of array elements of simple data types. These tools, however, suggest that if a single element of the array is initialized, then the entire array is considered to be initialized. Obviously, there is room for improvement.
- None of the tools address detection of complex, global, or interprocedural uninitialized array variables.
- All the tools detected basic buffer overflow and format string vulnerabilities

Please note that by combining the information in table 1 and table 3, Klocwork identified more of the security rules than the other tools even prior to Motorola's engagement with them. The significance of this activity is that one must be aware of the relevant security rules applying to their domain before engaging any static analysis tool. This engagement is not sufficient with simple tool usage but must be significantly extended for product security. Most of the static analysis tools support extension capability, and the tools X, Y, Z also support extensions. Since Klocwork fared better in comparison with the other tools, the extension capability of the other tools was not studied in depth.

Opportunities

In Motorola's experience, use of static analysis tools has helped identify and correct a significant number of vulnerabilities in the coding phase. However, beyond the coverage of test cases indicated in this paper, there remain some opportunities for improvement for static analysis tools. The first opportunity is the considerable analysis required to prune the outputs of false positives. While all of the tools allow some means to minimize the effect, more effort is required. Secondly, the implementation of the checkers is typically example driven. As a result, the checker implementation can be only as complete as the set of examples. This creates the potential for false negatives. Finally, even though a relatively large range of memory related errors including memory leaks are reported by static analysis tools, there is still a need to run dynamic analysis tools for things like memory leak detection [3]. It would be a great benefit, if there could be improved techniques for memory leak detection and other memory related errors in static analysis tools. This type of capability could save a lot of time, effort and cost for software development organizations. Even the creation of an exhaustive test suite of memory related errors with a comparison of the popular static and dynamic analysis tools ability to detect all the different types of memory errors would be a big step forward.

In one open source code implementation of the https protocol, three high severity errors were identified in the original code and 17 high severity errors were identified in internally modified code. These errors related to security were detected, by running the code through Klocwork *Inforce* tool with the security options enabled. This example demonstrates the need for usage of such a tool for third party software as well as for internally developed software.

Recommendation

The paper thus far may read as a white paper for Klocwork. The value of this paper is in the approach used to make the software developed within our organizations better from both a quality and security perspective. First of all, it is important for an organization to take responsibility for the security of its software instead of relying on external security mechanisms. Secondly, in response to a significant number of security vulnerabilities in the coding phase, it is highly recommended to identify coding standards for the organization to follow. These coding standards can be reinforced through training and inspection. However, to optimize the return from these coding standards, they should be automated where possible. Our experience demonstrates that the commercial static analysis tools are lacking in a number of important security areas. It is absolutely necessary for people to own the security requirements for their static analysis tools and work with the vendor to enhance the capabilities. The tools lag the known concepts behind secure practices. Finally, one has to combine automated methods with manual methods such as inspection to capture as many of the errors as possible.

Conclusion

In this paper, the Motorola experience and approach in bringing a security focus to the coding phase has been shared, especially the use of static analysis tools for security. External experts in the security field were engaged for training and process enrichment. In particular, the coding standards were enhanced with security rules. After implementing the security enhanced coding standards, supporting these new standards in a static analysis tool became a major focus area. A majority of project teams in Motorola were already using the Klocwork tools for quality, which was a major factor in our use of this particular tool. However, a good percentage of these security rules were not detected by the tool. The vendor agreed to work with Motorola to improve the detection of violations to the security rules in the code and the related work has been described in this paper. The results from a couple of benchmarking exercises are also presented. A test suite published from MIT on buffer overflow, was used to identify and close the identified gaps in the Klocwork *Inforce* tool. In another study reported in this paper, popular static analysis tools were evaluated based on their support for the security rules in the Motorola coding standards. Based on this experience with static analysis tools, several opportunities for improvement in use of this technology were identified.

References

[1] Kratkiewicz, K. J. (May, 2005). *Diagnostic Test Suite for Evaluating Buffer Overflow Detection Tools – the companion test suite for "Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*. Retrieved September 11, 2007 from http://www.ll.mit.edu/IST/pubs/KratkiewiczThesis.pdf

[2] Howard, M., and LeBlanc, D. (2003). *Writing Secure Code*. Redmond, Washington: Microsoft Press.

[3] Wikipedia (N.D.) *Definition of Memory Leak* . Retrieved September 11, 2007 from <http://en.wikipedia.org/wiki/Memory_leak>

[4] Software Diagnotics and Conformance Testing Division. (July 2005*) SAMATE- Software Assurance Metrics and Tool Evaluation.* Retrieved September 11, 2007 from http://samate.nist.gov/index.php/Main_Page

Appendix A

C Secure Coding Standard	**Test** Case	**KW 6.1**	**KW 7.5**	**X**	**Y**	**Z**
Memory Allocation Check	1	X	X	X		
"Large" Arrays Flagged	2	X	EC			X
Access control for sensitive variables			EC			
Special characters in filenames or variables	4	X	EC			
Safe directories for file access			EC			
Check permissions prior to access filenames	18		EC		X	
User rights checked for file access			EC			
Null termination of string buffers	12		X			
Check return codes of library functions	5		EC		X	
%n substitution	15		EC			
Reference uninitialized variable	6	X	X			X
Unsafe library functions	7	X	X	X	X	X
Check variable lengths in unsafe functions	19	X	X	X	X	
Integer Overflow						
For addition and multiplication, result should not exceed operands	9					
Expressions as function call parameters	20		EC			
Buffer Overflow	10	X	X	X	X	X
Element references within array bounds	11	X	X	X		X

X indicates the coding standard is covered in the native code. EC indicates that the coding standard is covered partially or completely by an extensibility checker.

C Secure Coding Standard, continued	**Test** Case	**KW 6.1**	**KW 7.5**	**X**	**Y**	**Z**
String manipulation arrays are null terminated	12		X	X	X	
Externally provided strings not to be used in format string	13		EC	X	X	X
%s for printf	14		X			

X indicates the coding standard is covered in the native code. EC indicates that the coding standard is covered partially or completely by an extensibility checker.

Appendix B

Java Secure Coding Standard	**Test** Case	**KW 6.1**	**KW 7.5**	**X**	**Y**	**Z**
Restrictive Security Policy			X			
Secure Class Loader		X	X			
Initialization of Objects	3	X	X			
Private classes, methods, variables	4	X	X			
Finalized classes and methods	5	X	X			
Class Cloning	7	X	X	X		
Serialization	8		X			
Undeserializeable Classes	9		X			
Static Field Variables	10	X	X			
Inner Classes and sensitive data	11		X			
Arrays and Strings with sensitive data	13					
Random Number Generator	14	X	X	X	X	
Class Comparison by Name	15	X	X			

Appendix C

C++ Secure Coding Standard	**Test Case**	**KW 6.1**	**KW 7.5**	**X**	**Y**	**Z**
Object Memory Allocation Check	1		EC			
I/O Streams in C-style strings	2		X			
C-style strings	3		EC			
Conversion to C-style strings	4		EC			
Throw/ “New” Operator	5		EC			
Initialize primitive types	6	X	X			X
Array Initialization	7	X	X			
Deleting with void pointers and objects w/children	9		EC			
Non-primitive array manipulation			EC			
Object Slicing`	10		EC			
Pointer casting	11					
C-style casting and static _cast	12a,b					
Delete[] for array	8					
Array use vectors	14		EC			
Safe accessor methods	15		EC			
Virtual destructor of base, polymorphic classes	17					
Auto pointers	31		EC			
Public method checks arguments	21					
Static member variables	22		EC			
Pointers to temporary objects	23		EC			
Exception handling	24		EC			
Exceptions throw objects and not pointers	26		EC			
Catch exceptions	27		EC			
Unhandled exceptions	28		EC			

Evaluation of Static Source Code Analyzers for Avionics Software Development

Redge Bartholomew
Rockwell Collins
rgbartho@rockwellcollins.com

Abstract

This paper describes an evaluation of static source code analyzers. The purpose of the evaluation was to determine their adequacy for use in developing real-time embedded software for aviation electronics where the use of development tools and methods is controlled by a federal regulatory agency. It describes the motivation for the evaluation, results, and conclusions.

1. Introduction

Business issues motivate avionics developers to accelerate software development using whatever tools and methods will reduce cycle time. At the same time the FAA requires that all software development tools and methods comply with RTCA DO-178B, which requires disciplined and rigorous processes that can also be time consuming and expensive. Source code reviews and structural coverage testing, for example, are both required, and both typically involve considerable manual effort. An obvious solution within the faster-cheaper-better spiral of continuous development improvement is automation: perform the required analysis, testing, and documentation using tools that replace inconsistent and expensive human actions with consistent and (comparatively) cheap machine actions.

A static source code analyzer is an example. Potentially, it could replace manual source code reviews, some of the structural coverage testing, enforce compliance with a project coding standard, and produce some of the required documentation. In addition, by eliminating a large number of latent errors earlier in the development cycle, it could significantly reduce down stream activities like unit, integration, and system test. The number of errors a static analyzer has found in open source software provides anecdotal support for this last possibility [8].

However, the use of a static analyzer for avionics development encounters an issue that standard desktop development typically does not. The development process, tools, and qualification plans as described in the Plan for Software Aspects of Certification (RTCA DO-178B, paragraph 11.1), or its referenced plans, must be approved by the FAA's Designated Engineering Representative. To be cost effective, static analysis tools might have to be accurate enough and cover a broad enough spectrum of errors that the FAA would allow their use to replace manual analysis: if full manual analysis is still required, the amount of effort its use eliminates may not be large enough to justify its acquisition and usage costs.

In addition, some appeared to scale poorly, some appeared to have high false positive rates, some seemed to have ambiguous error annunciation markers, and some appeared to integrate poorly into common development environments. These issues could significantly reduce any benefit resulting from use.

This paper describes an internal evaluation of static source code analyzers. It had 3 objectives: to determine if static source code analyzers have become cost effective for avionics software development; to determine if they can be qualified to reduce software developers verification work-load; and to determine conditions under which avionics software developers might use them. There was no effort to determine down-select candidates or a source-selection candidate, nor was there any effort to achieve statistically significant results. It provided input to a decision gate in advance of a proposed pilot project.

In addition, it relied on software subject to publication restriction and on information subject to proprietary information exchange agreements. As a result, neither product names nor vendor names are identified. Information that could imply product or vendor has been withheld.

2. Background and Scope

2.1 Effectiveness

One concern over the use of a static analyzer was whether it could reduce down-stream development costs by reducing rework: whether tools can detect kinds of errors and quantities of errors that manual code reviews typically do not, that typically escape into downstream development and maintenance phases.

Another concern was whether they could reduce the cost of compliance with verification standards by automating manual source code reviews, or parts of them. If a static analyzer could be qualified against specific classes of errors (e.g., stack usage, arithmetic overflow, resource contention, and so on) and shown accurate above some minimum threshold, then it might be possible to eliminate those error classes from the manual reviews. Instead, an artifact could be submitted demonstrating that the check for those error classes was performed automatically by a qualified tool. Error classes not included in the tool qualification would still be subject to manual review.

A final concern was whether the use of an unqualified static analyzer as a supplement to a manual review would increase the number of downstream errors. If a static analyzer is effective against some kinds of errors but not others – e.g., catches pointer errors but not arithmetic overflows – this could be the case. Code reviewers could assume static analyzers are equally effective against all classes of errors, and minimize the effort they put into the manual analysis.

2.2 Scope

Time and resources did not allow for a determination of average number of errors detected by manual source code review versus those detected by a static analyzer. There is enough anecdotal evidence of errors escaping manual review into the test phase to suggest that a static analyzer will typically detect quantities of errors within a targeted error category that manual review will not. A comparison of effectiveness between manual and automated analysis limited to the error categories within which a given tool was designed to operate is a logical next step.

Currently, vendors advertise software security and software assurance capabilities. Security in the static analysis context appears to signify resistance to cyber attack and focuses on detection and elimination of errors that in the past have frequently been exploited to deny or corrupt service (e.g., buffer overflow). Assurance appears to signify the detection of the broad band of unintended latent errors whose detection in the real-time embedded context is the subject of source code walkthroughs and structural analysis testing. There appeared to be no error set that could distinguish assurance from security, nor did there appear to be separate static analysis products exclusive to assurance versus security. The focus of this evaluation was software assurance.

.

3. Method

The evaluation included 18 different static source code analyzers. Because of resource constraints only 6 were evaluated in house. There are, however, published comparative evaluations, and these were used as a supplement. The criteria chosen for the internal evaluations are common with most of the published evaluations.

To account for differences in standards across the different reviewers, results were normalized. Tools evaluated by more than a single source provided scale calibration points across the external evaluation sources [1-5] using the internal evaluations as the benchmark.

Some vendors do not provide evaluation licenses. As a consequence, some of the in-house evaluations were the result of vendor-performed demonstrations. Some were the result of web-based demonstrations hosted on the vendor's web site. In these cases vendors provided some of the performance data, but some of it resulted from interpolating available results (e.g., if the tool provided buffer overflow detection for a standard data type, then with confirmation from the vendor, it was assumed the tool provided buffer overflow detection for all standard data types).

4. Evaluation Criteria and Methodology

Based on input from developers, engineering managers, and program managers, the criteria used for the evaluation were analysis accuracy, remediation advice, false positive suppression, rule extension, user interface, and ease of integration with an Integrated Development Environment (IDE - e.g., Eclipse). Analysis accuracy consisted of the detection rate for true positive (correctly detected errors), true negative (correctly detected absence of errors), false positive (incorrectly detected errors), and false negative (incorrectly detected absence of errors). Remediation advice is the information the tool provides when detecting an error – information that allows the developer to better understand the error and better understand how to eliminate it. False positive suppression is a measure of how easy it is to suppress redundant error, warning, and information messages or the extent to which a tool allows suppression. Rule extension is a measure of the extent to which the tool allows for the addition of error or conformance checks and how easily this is done. The user interface is a measure of how easy it was to learn to use the tool and how easy it was to perform common tasks. Price was not included, primarily because vendor prices vary greatly depending on quantity, other purchased products, and marketing strategy.

The evaluation criteria were weighted. Potential users and their management felt analysis accuracy was the most important tool attribute so its weight was arbitrarily set at twice that of both remediation advice and false-positive suppression, and 3 times that of IDE integration and user interface. Based on developer input, the importance of rule extensibility was arbitrarily set at half that of IDE integration and user interface. Internal evaluations used a 3 point scale, where 3 was best and 0 indicated the absence of a capability. The published comparative evaluations each used a different scoring mechanism. Their results were normalized.

To the extent vendors provided evaluation licenses, analysis accuracy was measured using a small subset of the NIST SAMATE Reference Dataset (SRD) test cases [10]. Additional test cases were created to fill gaps (e.g., tests for infinite loops). Evaluations were limited to 6 general error categories: looping, numeric, pointer, range, resource management (e.g., initialization), and type. In all, 90 test cases were used. Both the downloaded test cases as well as those written for this evaluation were written in C to run on a Wintel platform against the MinGW 3.1.0 gcc compiler.

Some vendors provided in-house demonstrations but were reluctant to analyze small code segments, preferring to demonstrate effectiveness against large systems or subsystems (i.e., > 500KSLOC). In those cases the evaluation team interviewed technical staff from the tool supplier to determine capability against specific error classes.

The group at MIT and MIT's Lincoln Lab evaluated 5 tools against a set of test cases that were subsequently submitted to the NIST for inclusion in the SRD. The DRDC evaluated tools against test cases that also were submitted to the NIST for inclusion in the SRD. In the case of the other published evaluations, the basis for the accuracy evaluation is unknown.

5. Results

The results of the evaluation are these: For C and C++, some static analyzers are cost effective; none of those evaluated could be qualified as a replacement for manual activities like source code reviews (and may even be detrimental as a supplement to them) or for structural coverage testing; but if used prudently, some can reduce the cost of implementation (code, test).

In general all evaluated tools displayed significant deficiencies in detecting source code errors against some of the error categories. Determining false negative thresholds of acceptability against the different error categories and then determining each tool's areas of acceptable strength and unacceptable weakness is a logical next step, but was outside the scope of this effort.

On the basis of performance against the criteria, tools fell into two tiers. Only two of the evaluated tools had good scores for analysis accuracy, user interface, remediation advice, and false positive suppression. In both cases rule extension/addition required separate products. One performed poorly against arithmetic, type transformation, and loop errors. Both scaled well from very small segments of code to very large systems. All things considered (e.g., installation, learning curve) both are most effectively used for system or subsystem error checking within the context of a daily/nightly automatic build process, as opposed to evaluating small daily code increments in isolation. Both tightly coupled error detection with change tracking. The change tracking feature could have a significant near-term impact on productivity (non-trivial learning curve) if integrated into an existing formal development process.

In the second tier, several had scores that ranged from very good to poor for analysis accuracy, remediation advice, rule extension, and false positive suppression. Several in this tier had good scores for error detection but poor scores for false positive rate and a cumbersome false positive suppression capability. Many in this tier did not scale well (up or down) – e.g., some of the evaluated versions crashed while analyzing large systems. Some had adequate accuracy and remediation advice once the large number of false positive messages was suppressed. Error analysis coverage is narrow compared to the two first tier tools – e.g., will not detect such C errors as:

```
char a[15];
strcpy(a, "0123456789abcdef");
```

or

```
int i = 2147483647;
i = i * 65536;
```

or

```
int i = 0;
while (i < 10)
{
        i = i - 1;
}
```

Nearly all vendors of error detection tools also provide conformance checking capabilities, usually via separate licensing. Those that provided licenses for this evaluation also provided rule-set extension capabilities. Although conformance checking comparisons were not part of this evaluation, in a brief review, most had adequate capability. Some had significant advantages over others - e.g., out-of-the-box rule set (MISRA C rule set already installed) and ease of extension and modification.

Finally, static analyzers that perform inter-procedural analysis provide capability not addressed by manual code reviews, which typically only address individual code units (e.g. single compilation units). The ability to detect errors resulting from the impact of cascading function and procedure calls is not realistically available to manual analysis but clearly advantageous, identifying errors that are usually detected during system test or operational test. It was convenient to run this kind of static analyzer as a part of the automatic system build.

6. Conclusions

Static analysis can cost-effectively reduce rework (detecting defects before they escape into downstream development phases or into delivered products) but currently cannot replace manual source code reviews. In general, they need better error detection accuracy and broader coverage across error classes.

6.1 Cost Effective Reduction of Rework

Some static analyzers – those with broad coverage and high accuracy – are simple enough to use and are accurate enough that downstream cost avoidance exceeds cost of use (license cost, cost of false positive resolution and suppression, etc.). Tools in this category detect some source code errors faster and more effectively than manual reviews. Development teams can use them informally, on a daily/nightly basis, throughout the implementation cycle (code and development test) when integrated into an automated build process, reducing cost by reducing the quantity of errors that escape into development testing, and by reducing the number of iterations through each test phase (e.g., unit, integration, functional, performance).

Static analyzers with more limited coverage and lower accuracy, internal demonstration of cost effectiveness is difficult. No one static analyzer was effective against very many of the error classes identified by the Common Weakness Enumeration [7]. In addition, within some error classes where detection capability existed, many demonstrated a high false negative rate against the limited number of test cases. Many with a low false negative rate had a high false positive rate. Distinguishing between true and false detections and suppressing the false positives was a significant effort. It was not clear that the less than optimal reduction in debugging and rework was enough to offset the increase in effort from false-positive analysis and suppression.

6.2 Reducing Formal Compliance Cost – Automated vs. Manual Analysis

Of the 18 static analysis tools evaluated, none was designed to detect all the kinds of errors manual analysis detects. Therefore, none could replace manual analysis in a development environment regulated by an industry standard like DO-178B.

Automation of a manual process like the code review would require qualification of the static analyzer: documented demonstration of compliance with requirements within the target operating environment to confirm the tool is at least as effective as the manual process it would replace (e.g., RTCA DO-178B, paragraph 12.2.2).

Currently, this would be difficult given the evolving status of the existing government and industry resources, and the absence of performance requirements. If an industry standard defined error categories (e.g., Common Weakness Enumeration), defined tests by which static analyzers could be evaluated against those categories (e.g., SAMATE Reference Dataset), and defined performance requirements against those tests (e.g., less than 1% false negative rate), compliant static analyzers might be able to eliminate manual review within targeted categories. Manual reviews would still be required, but detection of qualified error categories could be eliminated from them.

It is unclear that the size of the current safety-critical market is large enough to motivate the tool developer's investment in qualification. Over time, however, resolving the cost of tool qualification could follow the same path as code coverage tools, where the tool developer now sells the deliverable qualification package or sells a qualification kit from which the user produces the qualification package. It is also possible, if individual tools achieve broad enough error coverage and high enough accuracy, that a user (or possibly a user consortium) may be motivated to qualify it. Qualification by a government-authorized lab could also become cost effective for either the tool developer or the tool user.

6.3 Potential for Increasing Rework Cost

There is resistance to using existing static analyzers as a supplement to manual source code reviews [6]. The National Academy of Sciences established the Committee on Certifiably Dependable Software Systems to determine the current state of certification in the dependable systems domain, with the goal of recommending areas for improvement. Its interim report contained a caution against tools that automated software development in the safety-critical context:

> "…processes such as those recommended in DO-178B have collateral impact, because even though they fail to address many important aspects of a critical development, they force attention to detail and self-reflection on the part of engineers, which results in the discovery and elimination of flaws beyond the purview of the process itself. Increasing automation and over-reliance on tools may actually jeopardize such collateral impact."[6]

Given that all evaluated tools exhibited major failures against some error categories, reducing the effort that goes into the manual review would lead to an increase in errors found during the test phase and an increase in errors found in delivered products. For that reason, many of the FAA's Designated Engineering Representatives are reluctant to approve the use of a static analyzer even as a supplement to manual analysis.

Until tools exhibit broader coverage and greater accuracy, their use for any aspect of the formal source code review process (RTCA DO-178B, paragraph 6.3.4.f) is probably premature.

6.4 Automating Conformance Checking

If conformance checking is the primary concern, and error detection a secondary issue, any of the evaluated tools would perform adequately without additional functional or performance capability. Some are easier to use out of the box than others, but all significantly reduce the effort of achieving conformance with a coding standard.

6.5 Automating Non-Critical Error Detection

In environments where cost is the driving factor and there are no tool qualification issues (e.g., there is no false negative rate requirement), any of the low-end tools could be used without additional functional or performance capability. They can be simple to acquire and simple to install, with an intuitive interface. If it is open-source or inexpensive (e.g., less than $500 acquisition fee per user with a 20% per year recurring fee) and easy to use, demonstrating that it catches some of the common implementation errors that typically escape into the integration and test phases (e.g., uninitialized stack variable, buffer overflow) may be enough to justify usage cost (false positive suppression, learning curve, modification of standard development process, etc.).

In a development environment where there is no previous experience with static analysis, using a low-end tool to demonstrate the cost effectiveness of the technology could be a means of subsequently justifying the upgrade to a more expensive and more capable high-end tool.

7. References

[1] Zitser, Lippman, Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code", ACM *Foundations of Software Engineering 12*, 2004, available at http://www.ll.mit.edu/IST/pubs/04_TestingStatic_Zitser.pdf

[2] Kratkiewicz, Lippmann, "A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools", *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, National Institute of Standards and Technology, February 2006, pp. 44-51

[3] Michaud, et al, "Verification Tools for Software Security Bugs", *Proceedings of the Static Analysis Summit*, National Institute of Standards and Technology, July 2006, available at http://samate.nist.gov/docs/

[4] Newsham, Chess, "ABM: A Prototype for Benchmarking Source Code Analyzers", *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, National Institute of Standards and Technology, February 2006, pp. 52-59

[5] Forristal, "Review: Source-Code Assessment Tools Kill Bugs Dead", *Secure Enterprise*, December 1, 2005, http://www.ouncelabs.com/secure_enterprise.html

[6] Committee on Certifiably Dependable Software Systems, *Software Certification and Dependability*, The National Academies Press, 2004, pp. 11-12

[7] Common Weakness Enumeration, http://cve.mitre.org/cwe/index.html#graphical

[8] Chelf, *Measuring Software Quality: A Study Of Open Source Software*, posted March 2006 at http://www.coverity.com/library/pdf/open_source_quality_report.pdf

[9] Software Considerations in Airborne Systems and Equipment Certification RTCA DO-178B, December 1, 1992

[10] SAMATE Reference Dataset, National Institute of Standards and Technology, http://samate.nist.gov/SRD/

Common Weakness Enumeration (CWE) Status Update

Robert A. Martin
MITRE Corporation
202 Burlington Road
Bedford, MA 01730
1-781-271-3001
ramartin@mitre.org

Sean Barnum
Cigital, Inc.
21351 Ridgetop Circle, Suite 400
Sterling, VA 20166
1-703-404-5762
sbarnum@cigital.com

ABSTRACT

This paper is a status update on the Common Weakness Enumeration (CWE) initiative [1], one of the efforts focused on improving the utility and effectiveness of code-based security assessment technology. As hoped, the CWE initiative has helped to dramatically accelerate the use of tool-based assurance arguments in reviewing software systems for security issues and invigorated the investigation of code implementation, design, and architecture issues with automation.

1. INTRODUCTION

As the threat from attacks against organizations shifts from the network, operating system, and large institutional applications to individual applications of all types, the need for assurance that each of the software products we acquire or develop are free of known types of security weaknesses has increased. High quality tools and services for finding security weaknesses in code are maturing but still address only a portion of the suspect areas. The question of which tool/service is appropriate/better for a particular job is hard to answer given the lack of structure and definition in the software product assessment industry.

As reported last year [2], there are several ongoing efforts working to resolve some of these shortcomings, including the Department of Homeland Security (DHS) National Cyber Security Division (NCSD) sponsored Software Assurance Metrics and Tool Evaluation (SAMATE) project [3] being led by the National Institute of Standards and Technology (NIST) and the Object Management Group (OMG) Software Assurance (SwA) Special Interest Group (SIG) [4].

Since that time, there has been related work started by the Other Working Group on Vulnerabilities (OWG-V) within the ISO/IEC Joint Technical Committee on Information Technology (JTC1) SubCommittee on Programming Languages (SC22) [5] as well as the new efforts at the SANS Institute to develop a national Secure Programming Skills Assessment (SPSA) examination [6] to help identify programmers knowledgeable in avoiding and correcting common software programming weaknesses, among others.

While all of these efforts continue to proceed within their stated goals and envisioned contributions, they all depend on the existence of common description of the underlying security weaknesses that can lead to exploitable vulnerabilities in software. Without such a common description, these efforts cannot move forward in a meaningful fashion or be aligned and integrated with each other to provide strategic value.

As stated last year, MITRE, with support from Cigital, Inc., is leading a large community of partners from industry, academia, and government to develop, review, use, and support a common weaknesses dictionary/encyclopedia that can be used by those looking for weaknesses in code, design, or architecture, those trying to develop secure application, as well as those teaching and training software developers about the code, design, or architecture weaknesses that they should avoid due to the security problems they can have on applications, systems, and networks.

This paper will outline the various accomplishments, avenues of investigation, and new activities being pursued within the CWE initiative.

2. COMMUNITY

Over the last year 6 additional organizations have agreed to contribute their intellectual property to the CWE initiative. Done under Non-Disclosure Agreements with MITRE which allow the merged collection of their individual contributions to be publicly shared in the CWE List, AppSIC, Grammatech, Palamida, Security Innovation, SofCheck, and SureLogic have joined the other 13 organizations that have formally agreed to contribute.

In addition to these sources, the CWE Community [7], numbering 46 organizations, is now also able to leverage the work, ideas, and contributions of researchers at Apple, Aspect Security, Booz Allen Hamilton, CERIAS/Purdue University, Codescan Labs, James Madison University, McAfee/Foundstone, Object Management Group, PolySpace Technologies, SANS Institute, and Semantic Designs, as well as any other interested parties that wish to come forward and contribute.

Over the next year we anticipate the formation of a formal CWE Editorial Board to help manage the evolution of the CWE content.

3. UPDATES

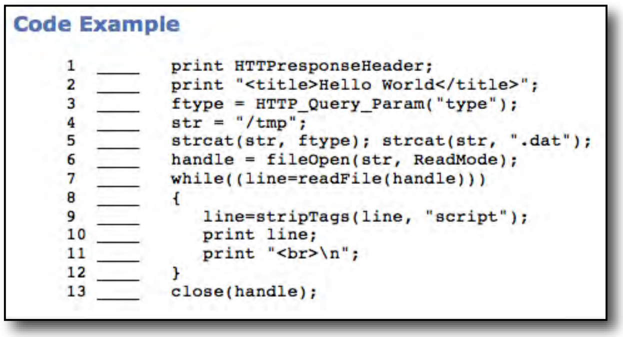
There were four drafts of CWE posted over the last year. With Drafts 4 and 5, CWE reached 550 and 599 items respectively. Draft 4 saw the introduction of the CWE ID field and Draft 5 included the introduction of predictable addresses for each CWE based on the CWE ID. During this timeframe the CWE web site expanded to include a “News” section, an “Upcoming Events” section, and a “Status Report” section. Draft 5 included additional details on node relations and alternate terms. With Draft 5 the CWE List was provided in several formats on the web site. Eventually this will be expanded upon to provide style-sheet driven views of the same underlying CWE XML content.

Draft 6 of CWE included a new category called “Deprecated” to allow duplicate CWEs to be removed by reassigning them to that category, which has a CWE ID but like the items it will hold, it is not part of totals for CWE. So there are 627 CWE IDs assigned with Draft 6, but two of the older CWEs have been moved to the new deprecated category, which also doesn’t count in the totals for CWE so there are 624 unique weakness concepts, including structuring concepts, in CWE Draft 6. The first formal draft of a schema for the core information that each CWE will have was finalized with Draft 6, covering the five groupings of information about each CWE, including “Identification”, “Descriptive”, “Scoping & Delimiting”, “Prescriptive”, and “Enhancing” types of information.

Draft 7 of CWE represents the first recipient of material from the CWE Scrub, described in section 5 of this paper. The main size type changes to CWE included the insertion of 7 new nodes to support grouping portions of CWE into additional Views and one CWE was deprecated. Further details of the changes in Draft 7 are included in the Scrubbing section of this paper.

4. VULNERABILITY THEORY

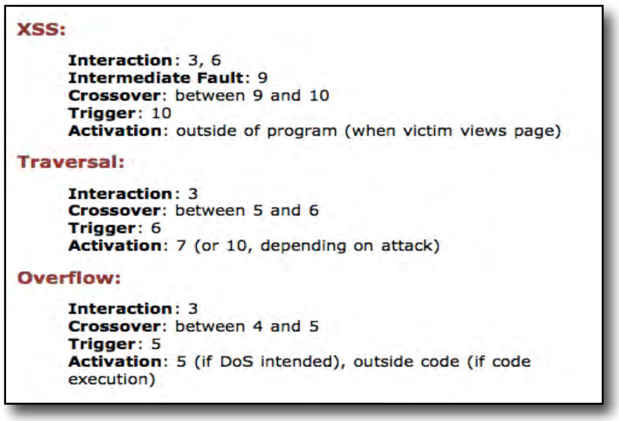
In parallel with the CWE content creation and as part of the Scrub activities, there has been considerable progress in documenting thoughts about the mechanics of vulnerabilities and how weaknesses, attacks, and environmental conditions combine to create exploitable vulnerabilities in software systems. Evolving the initial work on this topic, covered in the Preliminary List of Vulnerability Examples for Researchers (PLOVER) effort [8] in 2005, it now includes the results of working with great variety of issues covered in CWE. In July 2007, the “Introduction to Vulnerability Theory” [9] was published along with a companion document “Structured CWE Descriptions” [9]. The latter, using terminology defined in Vulnerability Theory, provides a first attempt to develop consistent descriptions of a broad and diverse set of over 30 CWEs. Given the many sources of information that CWE has combined it is important that we carefully comb through CWE to clarify and harmonize the use of terms and concepts. Another use of the terminology in Vulnerability Theory has been in annotating exemplar code. While common practice, the use of code snippets to show incorrect coding constructs is often hard to comprehend. The labeling of the individual artifacts within the code that are involved in a weakness shows great promise. For example, Figure 1 shows a code snippet that can be used to demonstrate three different types of CWEs, Cross-Site Scripting (79), Directory Traversal (22), and Unbounded Transfer (‘classic overflow’) (120).


Code Example

```
1  ____  print HTTPresponseHeader;
2  ____  print "<title>Hello World</title>";
3  ____  ftype = HTTP_Query_Param("type");
4  ____  str = "/tmp";
5  ____  strcat(str, ftype); strcat(str, ".dat");
6  ____  handle = fileOpen(str, ReadMode);
7  ____  while((line=readFile(handle)))
8  ____  {
9  ____     line=stripTags(line, "script");
10 ____     print line;
11 ____     print "<br>\n";
12 ____  }
13 ____  close(handle);
```

Figure 1: Code Example for CWEs 79, 22, and 120.

But without additional guidance it is difficult to identify the actual problem areas. Figure 2, specifying the line numbers involved in the different Vulnerability Theory concepts makes the discussion more precise by labeling the artifact concepts.


XSS:

Interaction: 3, 6
Intermediate Fault: 9
Crossover: between 9 and 10
Trigger: 10
Activation: outside of program (when victim views page)

Traversal:

Interaction: 3
Crossover: between 5 and 6
Trigger: 6
Activation: 7 (or 10, depending on attack)

Overflow:

Interaction: 3
Crossover: between 4 and 5
Trigger: 5
Activation: 5 (if DoS intended), outside code (if code execution)

Figure 2: Vulnerability Theory Artifact Labels for Code Example

5. SCRUBBING

The latest update to CWE included materials resulting from the “Scrubbing of CWE”. As mentioned earlier, CWE has material from many sources and covers a very wide range of concepts from a variety of perspectives and with varying levels of abstraction. Additionally, the software security industry has a long tradition of mixing together discussions of weaknesses, attacks, and results of exploited vulnerabilities that need to be addressed. Confronting these issues before CWE gets any larger is the purpose of the “CWE scrub”.

Resolving the mixed language issues blending description of weaknesses and attacks will involve two primary actions. The first is a rewriting of attack-centric language to clearly describe the underlying weakness that the attack is targeting. The second action will be an integration of related attack pattern references into the CWE schema and content such that the CWE can be effectively aligned to the Common Attack Pattern Enumeration and Classification (CAPEC) [10]. This part of the scrub effort is still in progress but it will not only improve clarity and consistency of the CWE weaknesses descriptions but will also add significant value to them by placing them within the context of how they are likely to be attacked.

Draft 7 includes over 800 major and minor changes due to the scrub. Examples of the changes include consistent naming conventions and spell checking at one end and revisiting the descriptions, relationships, context notes, and examples on the other. A detailed delta report capturing the change between Draft 6 and Draft 7 will be available on the CWE web site.

In addition to the above items, our study of the concepts captured in CWE has led us to identifying additional ways of capturing and describing the mechanics of the weaknesses, as described in Section 4. Two examples of concepts that CWE needs to be able to capture include series of weaknesses and simultaneous weaknesses. Figure 3 illustrates a series of weaknesses could be something like how an incorrect range check allows an integer overflow to occur which then leads to insufficient memory allocation which allows a heap overflow that could lead to code execution or a memory corruption induced crash. A simultaneous set of weaknesses could be an incorrect input cleansing, along with a guessable file name/path, and incorrect permissions that allows an attacker to download a sensitive file. CWE needs to be able to capture both of these types of situations within the XML data about the different weaknesses.

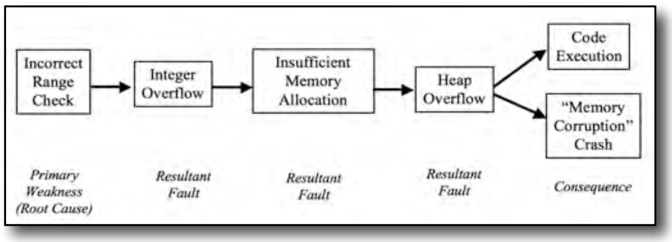


Figure 3: A Weakness Series

Another area that needs to be addressed in the CWE Scrub is the levels of abstraction CWE will support. In Draft 6 there were 466 CWEs that have no child nodes and are thought to be lowest-level concepts of a weakness (referred to as a vulnerability type). However, there are 44 other nodes that many consider to be lowest-level vulnerability types yet they have child nodes in CWE. Some of the lowest level concept child nodes are things that a static code analysis would not be able to recognize yet a dynamic code testing approach would. Additionally it would appear that these low level child nodes would be helpful concepts for developers, testers, and project managers to understand, make use of, and work with. So maybe the different uses of CWE should be supported by an ability to project parts of CWE and CWE needs a "sub-type" concept for these items? Similar issues surround the categories that CWE uses to explain hierarchical relationships amongst CWE items. There are currently 105 categories with child nodes associated to them. However there are another 9 categories that have no child nodes currently. Linking these categories to the appropriate child nodes or creating new child nodes is another topic of the CWE scrub.

To support the scrub activity and the ongoing review and enhancement of CWE a publicly archived CWE-RESEARCH-LIST has been set up and appropriate software security researchers were invited to sign-up. At the writing of this paper there are over a hundred subscribers already but everyone interested in the direction and evolution of CWE is encouraged to join the list and participate. Similarly a Research area has been established on the CWE web site with background information about the current and evolving ideas for scrubbing CWE as well as Use Cases and Stakeholder analysis.

6. COMPATIBILITY & EFFECTIVENESS

About a month after Draft 5 of CWE was posted the CWE Compatibility and Effectiveness Program was announced. CWE Compatibility is focused on the correct use of CWE Ids by tools and services while CWE Effectiveness is focused on determining which tools and services are effective in finding specific CWEs. The CWE Compatibility and Effectiveness Program [11] provides for a product or service to be reviewed and registered as officially "CWE-Compatible" and "CWE-Effective," thereby assisting organizations in their selection and evaluation of tools and/or services for assessing their acquired software for known types of weaknesses and flaws, for learning about the various weaknesses and their possible impact, or to obtain training and education about these issues. Detailed requirements defining CWE Compatibility and Effectiveness can be found on the CWE web site.

Currently, 12 organizations have declared that a total of 22 products & services are or will be CWE Compatible. This includes capabilities from Fortify Software, GrammaTech Inc., Armorize Technologies Inc., Klocwork Inc., CERIAS/Purdue University, Cigital Inc., SofCheck Inc., HP/SPI Dynamics, Ounce Labs, SANS Institute, Veracode Inc., and IBM/Watchfire. For a current list of CWE Compatibility see the CWE web site.

7. OUTREACH AND EDUCATION

A key component of any standardization effort that will be adopted and used by organizations is educating and informing all types of people about the effort, its motivation, plans, and potential impacts. Otherwise the standard may become an academic exercise that was never challenged to deal with practical usage cases by tool developers or users. The feedback from knowledgeable sources and the criticism/suggestions from those deeply involved and/or invested in the technologies, problems, or processes involved in an area being standardized is critical to identifying and rectifying any gaps or disconnects.

For the CWE initiative the Software Assurance Working Group meetings and Forums, co-sponsored by the Department of Defense (DoD) and the DHS, have been a major source of this type of interaction. Additionally, CWE has been presented in talks and discussions at conferences and workshops like the Tactical IA Conference, the InfoSecWorld Conference, the RSA Conference, the OMG Software Assurance Workshop, the main Black Hat Conference and the Black Hat D.C. Conference, the Defense Intelligence Agency Software Assurance Workshop, the GFIRST Conference, the Software and Systems Technology Conference, along with the IA in the Pacific Conference, and the AusCERT 2007 Conference.

Additionally, a paper on CWE was published for Black Hat D.C. [12] and an article was written for CrossTalk Magazine, the Journal of Defense Software Engineering [13].

8. CURRENT THOUGHTS ON IMPACT AND TRANSITION OPPORTUNITIES

As stated in the original concept paper that laid out the case for developing the CWE List [14], the completion of this effort will yield consequences of three types: direct impact and value, alignment with and support of other existing efforts, and enabling of new follow-on efforts to provide value that is not currently being pursued. Steady progress is being made to address and leverage each of the opportunities identified in that document.

Additionally, there have been three areas where CWE has been adopted fairly early that have great promise for spreading knowledge of CWE very quickly. The first is the use of CWE as a standard reference in the Open Web Application Security Project (OWASP) Top Ten Most Critical Web Application Security Vulnerabilities 2007 [15], the second is the creation and distribution by the SANS Institute, of a SANS CWE Poster, shown in draft form in Figure 4, and the third is the inclusion of CWE identifiers in the National Vulnerability Database (NVD) [16], shown in Figure 5, as refinement of the Vulnerability Type information they provide for publicly known vulnerabilities in packaged software.

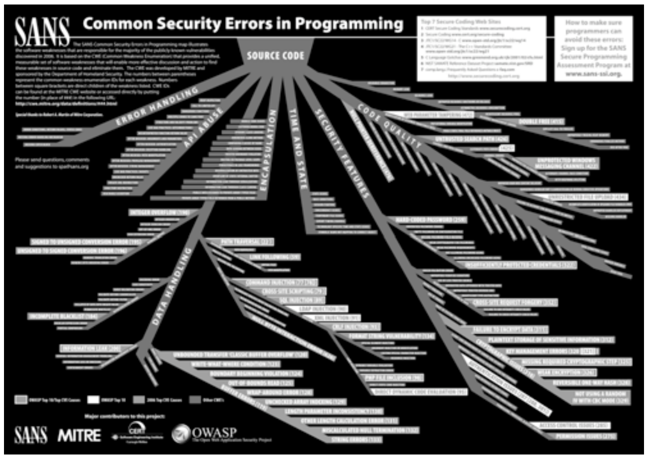
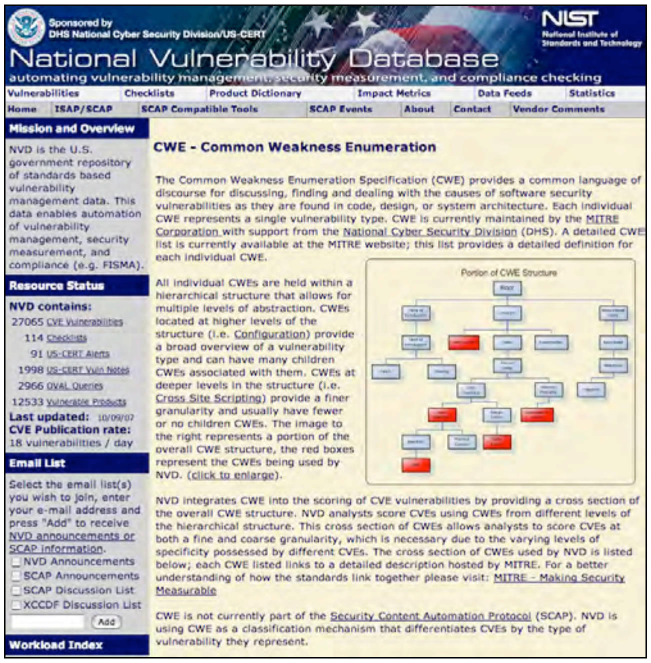


Figure 4: Draft of SANS CWE Poster.



Mission and Overview

NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

Resource Status

NVD contains:
27065 CVE Vulnerabilities
114 Checklists
91 US-CERT Alerts
1998 US-CERT Vuln Notes
2966 OVAL Queries
12533 Vulnerable Products
Last updated: 10/09/07
CVE Publication rate:
18 vulnerabilities / day

Email List

Select the email list(s) you wish to join, enter your e-mail address and press "Add" to receive NVD announcements or SCAP information.
- NVD Announcements
- SCAP Announcements
- SCAP Discussion List
- XCCDF Discussion List

CWE - Common Weakness Enumeration

The Common Weakness Enumeration Specification (CWE) provides a common language of discourse for discussing, finding and dealing with the causes of software security vulnerabilities as they are found in code, design, or system architecture. Each individual CWE represents a single vulnerability type. CWE is currently maintained by the MITRE Corporation with support from the National Cyber Security Division (DHS). A detailed CWE list is currently available at the MITRE website; this list provides a detailed definition for each individual CWE.

All individual CWEs are held within a hierarchical structure that allows for multiple levels of abstraction. CWEs located at higher levels of the structure (i.e. Configuration) provide a broad overview of a vulnerability type and can have many children CWEs associated with them. CWEs at deeper levels in the structure (i.e. Cross Site Scripting) provide a finer granularity and usually have fewer or no children CWEs. The image to the right represents a portion of the overall CWE structure, the red boxes represent the CWEs being used by NVD. (click to enlarge).

NVD integrates CWE into the scoring of CVE vulnerabilities by providing a cross section of the overall CWE structure. NVD analysts score CVEs using CWEs from different levels of the hierarchical structure. This cross section of CWEs allows analysts to score CVEs at both a fine and coarse granularity, which is necessary due to the varying levels of specificity possessed by different CVEs. The cross section of CWEs used by NVD is listed below; each CWE listed links to a detailed description hosted by MITRE. For a better understanding of how the standards link together please visit: MITRE - Making Security Measurable

CWE is not currently part of the Security Content Automation Protocol (SCAP). NVD is using CWE as a classification mechanism that differentiates CVEs by the type of vulnerability they represent.

Figure 5: NVD Use Of CWE.

Leveraging of the OMG technologies to articulate formal, machine parsable definitions of the CWEs to support analysis of applications within the OMG standards-based tools and models is continuing through an effort to create formalized CWE definitions. This effort, in conjunction with OMG standards-based modeling and automated code generation from models efforts and the NIST SAMATE Reference Dataset repositories creation continue to move forward and should create some very promising results. Any tool/service capability measurement framework that uses the tests provided by the SAMATE Reference Dataset will be able to leverage the common weakness dictionary as the core layer of the framework.

Through all of these activities, CWE continues to help shape and mature the code security assessment industry, and dramatically accelerate the use and utility of these capabilities for organizations and the software systems they acquire, develop, and use.

9. ACKNOWLEDGMENTS

The work contained in this paper was funded by DHS NCSD.

10. REFERENCES

[1] "The Common Weakness Enumeration (CWE) Initiative", MITRE Corporation, (http://cwe.mitre.org/).

[2] Martin, R., Barnum, S., "A Status Update: The Common Weaknesses Enumeration". Proceedings of the Static Analysis Summit, NIST Special Publication 500-262, July 2006.

[3] "The Software Assurance Metrics and Tool Evaluation (SAMATE) project", National Institute of Science and Technology (NIST), (http://samate.nist.gov).

[4] "The OMG Software Assurance (SwA) Special Interest Group", (http://swa.omg.org).

[5] "ISO/IEC JTC 1/SC22/ Other Working Group: Vulnerabilities", ISO/IEC JTC 1/SC 22 Secretariat, (http://www.aitcnet.org/isai/).

[6] "SANS Software Security Institute", SANS Institute, (http://www.sans-ssi.org/).

[7] "The Common Weakness Enumeration (CWE) Community", MITRE Corporation, (http://cwe.mitre.org/community/).

[8] "The Preliminary List Of Vulnerability Examples for Researchers (PLOVER)", MITRE Corporation, (http://cve.mitre.org/docs/plover/).

[9] "Introduction to Vulnerability Theory" and "Structured CWE Descriptions Documents", MITRE Corporation, (http://cwe.mitre.org/about/documents.html).

[10] The Common Attack Pattern Enumeration and Classification (CAPEC) Initiative", Cigital, Inc. and MITRE Corporation, (http://capec.mitre.org/).

[11] "The Common Weakness Enumeration (CWE) Compatibility Declarations", MITRE Corporation, (http://cwe.mitre.org/compatible/organizations.html).

[12] Martin, R. A., Christey, S., "Being Explicit About Software Weaknesses". "Black Hat DC Training 2007, " February, 2007 Arlington, VA.

[13] Martin, R. A., "Being Explicit About Security Weaknesses". "CrossTalk: The Journal of Defense Software Engineering", (http://www.stsc.hill.af.mil/CrossTalk/2007/03/), March 2007.

[14] Martin, R. A., Christey, S., Jarzombek, J., "The Case for Common Flaw Enumeration". "NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics", November, 2005 Long Beach, CA.

[15] "OWASP Top Ten Project 2007", Open Web Application Security Project, (http://www.owasp.org/index.php/Top_10_2007).

[16] "National Vulnerability Database (NVD) ", National Institute of Science and Technology (NIST), (http://nvd.nist.gov/nvd.cfm).

A Practical Approach to Formal Software Verification by Static Analysis[*]

[Extended Abstract]

Arnaud Venet
Kestrel Technology LLC
4984 El Camino Real #230
Los Altos, CA 94022
arnaud@kestreltechnology.com

ABSTRACT

Static analysis by Abstract Interpretation is a promising way for conducting formal verification of large software applications. In spite of recent successes in the verification of aerospace codes, this approach has limited industrial applicability due to the level of expertise required to engineer static analyzers. In this paper we investigate a pragmatic approach that consists of focusing on the most critical components of the application first. In this approach the user provides a description of the usage of functionalities in the critical component via a simple specification language, which is used to drive a fully automated static analysis engine. We present experimental results of the application of this approach to the verification of absence of buffer overflows in a critical library of the OpenSSH distribution.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*

Keywords

Static analysis, abstract interpretation, formal verification, buffer overflow

1. INTRODUCTION

The term static analysis is most often employed for denoting the detection of software errors or vulnerabilities by automatic inspection of source code. Static analysis tools in this category–like those commercialized by Coverity [1] or Klocwork [2] to name a few–have become increasingly popular among developers and enjoy widespread use in the software industry. However, this form of static analysis can only point to defects in the code but does not guarantee that all have been found, even if only a single class of defects is considered, like buffer overflows. A static analysis technology called Abstract Interpretation [8, 9] can make stronger claims for certain classes of software defects. The validity of such claims is backed by a rigorous mathematical theory underpinning the implementation of the static analyzer. Decidability issues are avoided by allowing the analyzer to give indeterminate results. These indeterminates are mere false positive most of the time but may also point to a real problem. The effectiveness of a static analyzer based on Abstract Interpretation is measured by its *precision* i.e., the ratio of false positives in the analyzer's output. A static analyzer that does not yield any false positive provides high assurance that the code is free of a certain class of defects.

Using static analysis to perform formal software verification sounds attractive at first: there is no need to build a model of the application, the verification process is fully automated and is conducted on the very code that will run on the target platform. However, the reality is somewhat disappointing. In order to achieve formal verification, the number of false positives produced by the analyzer must be zero or at least very small. Reaching this level of precision on real software systems requires (1) a substantial amount of work tuning the analysis engine, and (2) an excellent knowledge of the context in which the application operates (input parameters, sensor data, interruptions, etc.). For example, the design of the ASTREE static analyzer [5, 10], which has been used to verify the correctness of floating-point arithmetic in the electric flight control code of the Airbus A380, monopolized the attention of six world-class experts in Abstract Interpretation during a couple of years. Getting rid of all false positives required devising highly sophisticated algorithms to handle the unique characteristics of this code e.g., a domain of ellipsoids to analyze linear digital filters [5] and a representation of inequalities for floating-point variables [12].

However, some false positives cannot be removed by just improving the analysis engine, since they require information on the operating environment of the program that is not present in the code. For example, in our past experience we had to analyze the attitude control system of a satellite. The analyzer that we were using performed well but turned up a number of false positives that resisted all our attempts to improve the precision of the algorithms. A careful investigation of the origin of these false positives revealed that they were all caused by the lack of information on the variable

[*]This work was funded by the OSD SBIR contract FA8750-06-C-0146.

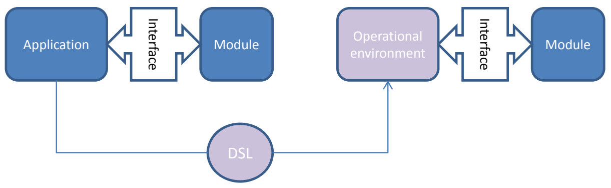


Figure 1: Modeling the usage of the module through its interface.

containing the altitude of the satellite. Simply adding the assertion that the altitude is always positive was sufficient to remove all remaining false positives.

This approach requires a close interaction between a group of experts in Abstract Interpretation and a group of experts of the application to be verified. It is difficult to imagine it being applied to a large variety of codes. The major bottleneck is the availability of experts in Abstract Interpretation who are willing to spend time on such projects. Although it is possible to build a general-purpose static analysis tool that exhibits good precision and performance in average–PolySpace Verifier is an example [4]–the number of indeterminates will still be too high for the purpose of high assurance. In this paper, we report on ongoing work for making Abstract Interpretation-based analyzers easier to use in practice without sacrificing too much precision. We are investigating a divide-and-conquer approach that allows application experts to use a generic static analysis engines on the most critical components of a software application. Our approach is described in Sect. 2. In Sect. 3 we describe an application of this approach to the verification of absence of buffer overflows in a critical library of the OpenSSH distribution.

2. DIVIDE-AND-CONQUER APPROACH

Abstract Interpretation is a well defined theory, which provides a systematic methodology for constructing sound static analyzers [7]. Static analyzers obtained by a straight application of the theoretical framework will not scale to hundreds of thousands of lines of code. Engineering scalable and sound static analyzers is extremely challenging and requires specializing the algorithms for the application or familiy of applications considered [13, 10]. However, if we limit the size of the programs to be analyzed to a few thousand lines, then it is possible to build a fairly general static analyzer that can handle a broad spectrum of programs for a given property (array-bound compliance, floating-point overflows, etc.) with high precision. We propose to apply such analyzers to small critical components of software applications. This approach is justified by the empirical observation that many large applications consist of a collection of smaller components. For example, in our previous work on NASA flight-control software of Mars missions [13, 6] we observed that the Mars Exploration Rovers mission control software is about half a million lines of code. However, it is made of over one hundred threads, each one acting as an independent unit and controlling either a particular instrument (like the high-gain antenna) or a phase in the mission (like entry-descent-landing). The monolithic structure of the electric flight control code of the A380, where interdependent operations may spread over hundreds of thousand lines of, is unique and mostly due to the fact that the code is automatically generated from higher level specifications.

This approach is viable if an application expert with a limited grasp of static analysis can successfully use an analyzer on a module of the application. As we previously observed, the operational environment of the applications is important for precision. Our approach adds another dimension to that problem, since we are now analyzing a module and the interactions between the module and the rest of the code have to be modeled. We assume that the module comes with a clearly defined interface. This is not an unrealistic assumption. For example, all threads in the Mars Exploration Rovers mission control software communicate using a common mechanism based on message queues, with a carefully specified format of messages. We propose to use a domain specific language (DSL) to model the interaction between the module and therest of the code through its interface as depicted in Fig. 1. The static analyzer takes as inputs the code of the module together with the model of its environment. The DSL provides a precise formal definition of the context in which the module is executed in a form that is easily intelligible by the user. We are currently working on a DSL for a family of representative program properties verifiable by static analysis.

We have chosen the OpenSSH 4.3 application bundle [3] as a realistic application for demonstrating the feasibility of our approach. The applications in OpenSSH use a common buffer library for the internal storage of data transmitted across the networks. This library implements dynamic buffers that transparently grow in order to fit the data stored therein. The buffer library is a critical component of the application bundle that is implemented using sophisticated algorithms. A buffer in OpenSSH is implemented in an object-oriented style, using a structure that contains information about the size of the buffer and the space available. The buffer structure is defined as follows:

```
typedef struct {
  u_char *buf;  /* Buffer for data. */
  u_int alloc;  /* Number of bytes allocated for data. */
  u_int offset; /* Offset of first byte containing data.*/
  u_int end;    /* Offset of last byte containing data. */
} Buffer;
```

The basic interface of the buffer library contains the following functions:

```
void  buffer_init(Buffer *);
void  buffer_clear(Buffer *);
void  buffer_free(Buffer *);

u_int buffer_len(Buffer *);
void *buffer_ptr(Buffer *);

void  buffer_append(Buffer *, const void *, u_int);
void *buffer_append_space(Buffer *, u_int);
```

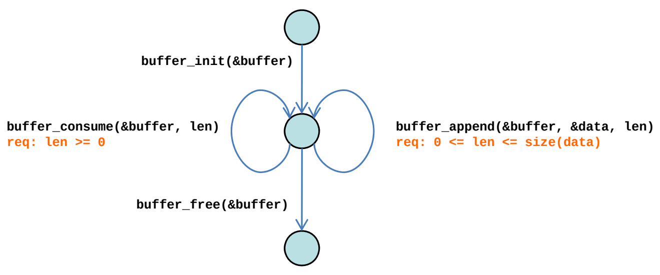

buffer_init(&buffer)
buffer_consume(&buffer, len)
req: len >= 0
buffer_append(&buffer, &data, len)
req: 0 <= len <= size(data)
buffer_free(&buffer)

Figure 2: Modeling the usage of the module through its interface.

```
void  buffer_get(Buffer *, void *, u_int);

void  buffer_consume(Buffer *, u_int);
void  buffer_consume_end(Buffer *, u_int);

void  buffer_dump(Buffer *);

int  buffer_get_ret(Buffer *, void *, u_int);
int  buffer_consume_ret(Buffer *, u_int);
int  buffer_consume_end_ret(Buffer *, u_int);
```

As previously described the user must supply a model of the interactions between the module and the rest of the system in order to enable the separate analysis of the module. The DSL that we have designed so far for modelling operational environments is based on state machines describing the order in which the functions in the interface are used, together with constraints on the arguments of the functions. Some of these functions store data into the buffer that are read from a byte array passed in the argument together with the number of bytes to read, like in the `buffer_append` function. Therefore, the interface requirement for such a function guaranteeing a proper use is that the number of bytes to read be less than or equal to the size of the byte array to read from. We have similar interface requirements for functions that read data from the buffer and store them in a byte array like `buffer_get`. Functions of the buffer library cannot be used in any order. A buffer must first be initialized using the `buffer_init` function, then any of the buffer manipulation functions can be applied in any order. The buffer is finalized and memory disposed of properly by the `buffer_free` function. Interface usage and function parameter requirements are illustrated in Fig. 2, where we depicted only a few functions for the sake of clarity.

3. CASE STUDY: OPENSSH'S BUFFER LIBRARY

The complexity of the implementation of the buffer library in OpenSSH comes from the fact that a buffer grows on demand, depending on the size of the data that are written into it. The growth of the buffer is controlled by the following function:

```
/*
 * Appends space to the buffer, expanding the buffer if
 * necessary. This does not actually copy the data into
 * the buffer, but instead returns a pointer to the
 * allocated region.
 */

void * buffer_append_space(Buffer *buffer, u_int len)
{
  u_int newlen;
  void *p;

  if (len > BUFFER_MAX_CHUNK)
    fatal("buffer_append_space: len %u not supported",
          len);

/* If the buffer is empty, start using it from the
   beginning. */
  if (buffer->offset == buffer->end) {
    buffer->offset = 0;
    buffer->end = 0;
  }
  restart:
/* If there is enough space to store all data, store it
   now. */
  if (buffer->end + len < buffer->alloc) {
    p = buffer->buf + buffer->end;
    buffer->end += len;
    return p;
  }
  /*
   * If the buffer is quite empty, but all data is at
   * the end, move the data to the beginning and retry.
   */
if(buffer->offset > MIN(buffer->alloc, BUFFER_MAX_CHUNK))
{
    memmove(buffer->buf, buffer->buf + buffer->offset,
    buffer->end - buffer->offset);
    buffer->end -= buffer->offset;
    buffer->offset = 0;
    goto restart;
  }
  /* Increase the size of the buffer and retry. */

  newlen = buffer->alloc + len + 32768;
  if (newlen > BUFFER_MAX_LEN)
    fatal("buffer_append_space: alloc %u not supported",
newlen);
  buffer->buf = xrealloc(buffer->buf, newlen);
  buffer->alloc = newlen;
  goto restart;
  /* NOTREACHED */
}
```

This function is quite complex and uses comparisons between the size of the data to store in the buffer and the available space to reallocate a buffer sufficiently large. Buffers are used throughout the OpenSSH distribution to store all data communicated through the network. They are the key data structure in the application and constitute an excellent example for our study. We want to verify that this implementation is not prone to buffer overflows.

We have developed a generic static analyzer for buffer overflows using Kestrel Technology's static analysis development platform CodeHawk. This analyzer features an efficient implementation of the polyhedral abstract domain [11] and optimized fixpoint interation algorithms. The analyzer is generic in the sense that it does not contain algorithms that deal with a particular code architecture. For example, consider the two following functions extracted from the buffer library:

```
/* Consumes the given number of bytes
   from the beginning of the buffer. */
```

```
int
buffer_consume_ret(Buffer *buffer, u_int bytes)
{
  if (bytes > buffer->end - buffer->offset) {
    error("buffer_consume_ret: trying to get more
           bytes than in buffer");
    return (-1);
  }
  buffer->offset += bytes;
  return (0);
}

void
buffer_consume(Buffer *buffer, u_int bytes)
{
  if (buffer_consume_ret(buffer, bytes) == -1)
    fatal("buffer_consume: buffer error");
}
```

In order to analyze the code precisely, the static analysis engine must be able to infer a correlation between the return value of function `buffer_consume_ret` and the invariant `bytes > buffer->end - buffer->offset`. The ASTREE analyzer handles a similar problem by using a special domain for finding correlations among Boolean and numerical variables [10]. In our case, the analyzer performs a sequence of interleaved forward and backward invariant propagations that achieves the same result. This algorithm is completely generic and may handle other forms of correlations among variables that are not necessarily Boolean. Note that this algorithm is not intended to scale to large codes, this is not our purpose here. We rather want a precise analyzer that can handle smaller codes with good precision without the need of manually fine-tuning the algorithms.

The buffer library contains 162 pointer checks that represent the safety conditions associated to each pointer operation in the library. The analysis runs in 35 seconds and is able to prove all 162 checks.

4. CONCLUSION

We have presented a divide-and-conquer approach that allows a user who is not an expert in static analysis to conduct formal software verification of small critical components of an application. Our approach is not compositional, in the sense that we do not verify the whole application by combining the results of individual components. Rather, we aim at providing a methodology and an accompanying toolset of fully automated static analyzers for verifying the most critical components of an application. The DSL will be helpful for saving environment models of previous analyses, in effect enabling development of libraries for reuse and giving clues for divide-and-conquer in other applications. We have successfully applied our approach to the verification of a complex critical module of the OpenSSH distribution. We vow to pursue these experiments and build a larger benchmark of open-source applications.

5. REFERENCES

[1] Coverity. `http://www.coverity.com`.

[2] Klocwork. `http://www.klocwork.com`.

[3] Open ssh. `http://www.openssh.org`.

[4] Polyspace verifier. `http://www.mathworks.com/products/polyspace`.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, June 7–14 2003.

[6] G. Brat and A. Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference*, 2005.

[7] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, New York, NY, 1979.

[10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, 2005.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, New York, NY, 1978.

[12] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[13] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 231–242, 2004.

Logical Foundation for Static Analysis: Application to binary static analysis for Security

Hassen Saïdi

Computer Science Laboratory
SRI International
saidi@csl.sri.com
http://www.csl.sri.com/users/saidi/

Abstract

Static analysis has emerged in recent years as an indispensable tool in software verification. Unlike deductive approaches to program verification, static analysis can only prove simple properties. Moreover, the myriad of static analysis tools employ specific techniques that target specific properties of specific programs. Static analysis holds the promise of complete automation, scalability, and handling larger classes of properties and larger classes of systems, but a significant gap exists between such a goal and current static analysis tools. We argue that a logical foundation for static analysis allows the construction of more powerful static analysis tools that are provably correct, extensible, and interoperable, and can guarantee more complex properties of complex systems. We address these challenges by proposing a tool-bus architecture that allows the combination of several static analysis tools and methods. The combination is achieved at the logical level using decision procedures that implement combination of theories. We discuss the application of such ideas to binary program analysis in the context of intrusion detection and malware analysis.

Keywords static analysis, abstract interpretation, logical interpretation, dynamic analysis, invariant generation

1. Introduction

Our ability to apply software analysis tools is constantly challenged by the increasing complexity of developed software systems. Static analysis has emerged as an indispensable tool in software verification. Static analysis is however at a cross road. Figure 1 illustrates how static analysis sits at the frontier between low cost approaches to assurance such as testing, typechecking and dynamic analysis, and more formal and less scalable approaches such as model checking and deductive approaches. We argue that static analysis holds the promise of maintaining the advantages of low cost, scalability and high bug coverage, while providing degrees of correctness that are often associated with the less scalable and more labor intensive deductive approaches.

In a recent study [13], it was shown that current state-of-the-art static analysis tools exhibit significant shortcomings. The study argues that static analysis tools employ a wide range of techniques and features with varying degrees of success making their evaluation for correctness challenging. That is, the criteria more commonly used to evaluate tools such as detection, accuracy, and scalability are not enough to evaluate the degree of dependability of the target system after analysis. It is necessary to extend static analysis tools in order to extract from the analysis enough evidence to support either confidence or distrust in the target's dependability.

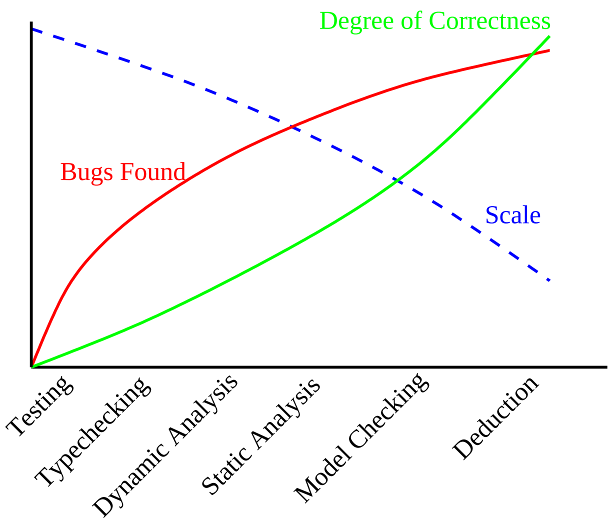


Figure 1. Formal Approaches

The shortcomings of current static analysis tools are summarized in the study in the following categories:

- Reliability: tools often handle specific programs and do not deal with all of the programming languages constructs and the dynamic run-time environment.
- Transparency: tools do not provide capabilities for extracting evidence that the flaws discovered by a static

analysis tool are genuine and supporting evidence on how the flaws can be triggered.

- Flaw Detection: tools have only a partial coverage of the space of potential flaws that can be addressed by static analysis. Achieving high coverage require using different tools with mixed results.
- Interoperability: tools tend to focus on one particular type of flaws and lack an infrastructure that allows them to share the result of their analysis.

The gap between the actual problems in practice and the tools and approaches requires addressing these shortcomings in the most general and effective way. We propose a tool-bus architecture that addresses these challenges by exploiting recent advances in static analysis techniques [8] and decision procedures and theorem proving [5]. Static analysis is not a trivial task even when source code is available. It becomes more challenging when reasoning about applications for which only the stripped binaries are available. We show how such a tool-bus architecture is applied to binary static analysis in a security context.

2. Logical Foundation for Static Analysis

Establishing the correctness of software is a computationally intractable problem in general. Software analysis typically is performed over a sound approximation or abstraction of the program's behavior. The choice of the abstraction or abstract domain determines the class of properties that the analysis focuses on. This approach is based on abstract interpretation [4] a technique for approximating programs behaviors. An abstract domain is represented by a lattice and the semantics of the program is captured by fixed point computations in the abstract lattice. A logical foundation for static analysis improves static analysis in the following ways:

- Formalization of fixed point computations provides proofs of soundness and completeness of the computation of the given static analysis tool. Using a theorem prover such as PVS [10], it is possible to reason about the correctness of fixed point computations as well as the combination of abstract domains.
- The combination of several abstract domains corresponds to the use of the corresponding tools in conjunction. However, In [8], it has been shown that the combination of abstract domains at the logical level, called logical interpretation, provides a more precise combination of abstract interpreters. Combining the abstract domains is performed by combination of theories corresponding to the abstract domains such as arithmetic and uninterpreted functions implemented in decision procedures such as Yices [5]. Logical theories can be used to define new and expressive logical lattices [12]. Abstract interpreters working over these logical lattices are constructed by using existing support for decision procedures over the corresponding logical theories. This support is available in the form of Satisfiability Modulo Theory (SMT) solvers, such as Yices. Logical lattices provide the perfect platform to balance expressiveness of abstract domains with their efficiency. Expressive logical lattices can be designed that can infer complex linear and nonlinear arithmetic invariants and even quantified invariants for arrays and heaps [9]. Using such expressive logical domains reduces the number of false alarms generated by a static analyzer. Logical lattices provide a uniform foundational framework to explore new logical domains and the limits and tradeoff between expressiveness and efficiency in building abstract interpreters.
- The interoperability between the different static analysis is achieved through assertions that are produced by each tool. Each tool can use an assertion that has been computed by another tool as an invariant that can aid its own computation and analysis. A single tool or technique will not be sufficient for achieving the levels of assurance desired. Multiple approaches, as depicted in Figure 1, need to be combined to work together to provide desired levels of security or safety. In this context, theorem proving technology will again be useful to achieve such an integration of different kinds of tools. A theorem proving engine, such as PVS, can provide the backbone required to carry out communication and transfer of results between types of analyzers.

With the exception of testing and dynamic analysis, all approaches in Figure 1 amount to computing an invariant of a program. That is an assertion that is true in all runs of the program. Each approach should be able to take advantage of invariants computed using other approaches. Examples of such combinations are deductive methods that use invariants generated using static analysis. Another example is finite abstractions computed using decision procedures such as the case of predicate abstraction [7] used in C code verification such as in the SLAM project [3]. In our application, we illustrate how runtime execution of a program generates interesting assertions about the program. Such assertions while true only for the observed runs can be checked using decision procedures in order to check if they hold for all possible runs of a program. If so, such assertions become invariants that can improve static analysis.

In other words, static analysis can greatly benefit from the advances in theorem proving and decision procedures to allow it to prove more complex properties and achieve a better coverage of potential flaws that will allow better assessment of the dependability of the target system. But it can also greatly benefit from any other technology that produces invariants that can then be consumed by the static analysis tools to improve their analysis. We illustrate this point in our binary static analysis by showing how static and dynamic analysis combined with theorem proving work better than any of these techniques alone.

3. Binary Static Analysis

Understanding what an executable does is paramount to the analysis of computer systems and networks in predicting accurately their behavior, and to the discovery of critical vulnerabilities that have devastating effect on our global computing infrastructure. Binary program analysis represents needs and challenges that are unmet by current analysis methods in general and static analysis in particular.

We are interested in two areas of research where binary program analysis is critical. The first one is in reverse engineering legitimate applications in order to predict their future behavior. Any deviation from such predicted behavior can be considered a malicious action that may trigger a diagnosis and a potential response. In such a case, binary program analysis provides us with a host-based intrusion detection capability with no false alarms that is applicable to a wide variety of applications from network services, to office applications. Using binary static analysis to build reliable models of applications will close the gap created by security tools that focus on publicly disclosed vulnerabilities that represent according to recent statistics no more than 7% of the total number of vulnerabilities in our computers. Binary static analysis does not only apply to the application code but extends to libraries for which the source code is unavailable. It has been shown that finding anomalies in the stream of system calls issued by user applications is an effective host-based intrusion detection capability, and static analysis is used to derive a model of application behavior resulting in a host-based intrusion detection system with three advantages: a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms. Therefore, static analysis produces a model in the form of an approximation of the behavior of the application. While this eliminates false alarms, since alarms are raised only when the application deviates from its over-approximation, it leads to a weakness of the model. That is, the model might accept sequences of system calls that the application would not allow. These sequences could potentially compromise the application and the underlying operating system. The more precise the model is, the less such attacks are possible. The precision of the model depends on the complexity of the application's code and the power of the static analysis employed to build the model. We observe that many types of attacks are preventable by making sure that the application models used for intrusion detection capture the semantics of the program in a simple and precise way. We observe that attacks always inevitably violate an invariant of the program, and that any approach to static-analysis-based intrusion detection will be weakened by the absence of such invariants from application models.

The second application of binary program analysis is malware detection and reverse engineering. Static binary analysis aims at reverse engineering the executable in order to answer the following fundamental questions: what is the intended behavior of the malware? how does it propagate itself? how does it protect itself from detection? is this a new instance of an already known malware, or does this malware contain logic that has never been observed before?. Current techniques for malware analysis rely on the execution of the malware and observing its behavior. However, a single or multiple executions of a malware instance does not provide a full picture of the potency of a malware and can only provide a partial image of its intended behavior due the multiple layers of obfuscation present in today's malware. Only static analysis can reveal the full extend of the malware behavior and its various triggers. Much of our malware analysis shares the same static analysis infrastructure with building models that represent the sequences of system calls and their arguments for arbitrary applications. Therefore, invariants generated at any control location will improve the construction of a more precise control flow graph of the program by taking into account the flow of data and context sensitivity captured using such invariants.

Source code analysis has seen significant advances in recent years. However, little has been achieved in analyzing binary programs. Most notable work in this area is the work of Value Set Analysis (VSA) [2] where the values of registers and memory addresses is approximated by a set of possible values determined statically. Our analysis focuses on invariant generation using a combination of static analysis, theorem proving in the form of decision procedure and dynamic analysis. We are interested in three main invariants of binary programs:

- linear invariants that represent constraints on function and system calls arguments and their return values and that determine the evaluation of jump conditions at any point of the program.
- stack invariants that represents constraints on the values of the stack at any point of the program
- heap invariants that represent constraints on the heap.

The three kinds of invariants can be expressed in the input language of Yices [5] that supports the usual types `int`, `real`, and `bool`, user defined recursive datatypes and bitvectors, as well as uninterpreted functions and dependent types.

4. Quasi-static Binary Analysis

We integrate static and dynamic analysis in a novel algorithm that we call quasi-static analysis in which constraints are generated dynamically from runs of the application as well as by our invariant generation techniques. We have implemented quasi-static analysis for C code in [11], and we describe in what follows how it is implemented for binary programs (Figure 2). There has been a wide body of literature that addresses the problem of invariant generation. What is important to notice is that with the help of decision procedures, it is much easier to prove that a given assertion is an invariant of a program than to discover such assertion using invariant generation techniques. For dynamic analysis of execution runs we use Daikon [6], a tool for generating likely invariants. What dynamic analysis with tools such as Daikon offers is an easy way of extracting assertions that are true for a particular run of the application. With the help of Yices [5] and similar decision procedures, we check if those assertions hold for all runs. Our Quasi-static analysis process relies on an initial phase of disassembly. We use IDA Pro [1] to generate from the executable and invoked DLLs, a control flow graph (CFG) from which further analysis is conducted. IDA pro is known to be the most effective disassembler tool.

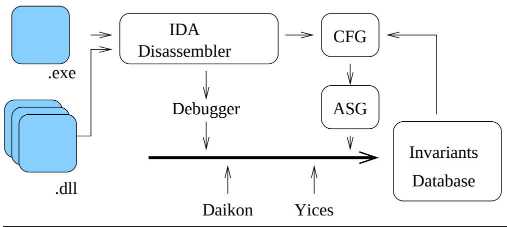


Figure 2. The Architecture of our Binary Analysis Tool-bus

4.1 Static Analysis

During the static analysis phase, we use the IDA Pro disassembler to obtain the CFG of the application and its DLLs. IDA Pro provide us with the a list of functions, memory locations containing constants such as strings, and the CFG for each individual function. It also identifies library calls and their arguments. The CFG is used by the static analyzer to compute invariants at each control location and to summarize individual functions in order to perform an interprocedural analysis and to build a global CFG. We use Yices to encode the semantics of instructions to define the logical lattice on which the analysis is performed. We also use Yices as an assertion checking engine for the assertions that are generated by dynamic analysis. An invariant database is used to store all of the computed invariants which are used to refine the global CFG. The global CFG can be viewed as an abstract state graph (ASG) similar to the one constructed by predicate abstraction [7]. Each newly generated invariant represents a refinement of the global CFG. The definition of the semantics of binary programs in Yices uses the usual Yices supported datatypes. The following is the Yices definition of the 32 bitvector type representing integers:

```
  (define-type int32 (bitvector 32))
```

We also define the stack type as a list as follows:

```
(define-type list
   (datatype nil
            (cons car::int32 cdr::list)))
```

The list is either empty (`nil`) or a list composed of a first element (`car`) of type 32 bit integer and a tail of type list (`cdr`). Consider the following simple example:

```
L1 :  mov   eax,    0
L2 :  mov   ebx,    eax
L3 :  inc   eax
L4 :
```

Each instruction is interpreted by its semantics that describe how the state variables are affected by the instruction. The registers, flags, and variables are all indexed by the number of the line of code where they appear:

- L2: eax_2 = 0
- L3: ebx_3 = eax_2 $\wedge$ eax_3 = eax_2
- L4: eax_4 = eax_3 + 1 $\wedge$ ebx_4 = ebx_3

On the other hand, each instruction generates an assertion about the program that might be propagated to other locations:

- L2: eax = 0
- L3: ebx = eax
- L4: eax = 1

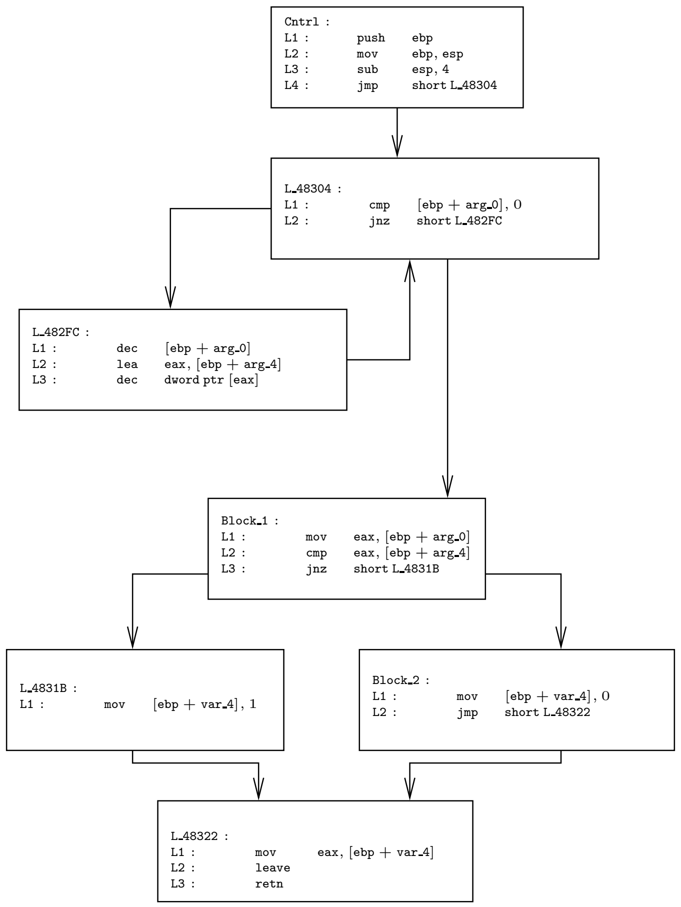
4.2 Dynamic Analysis

Our Dynamic analysis consists of tracing the application's execution within the IDA Pro debugger in order to extract runtime information. In particular, we are interested in register values, stack, and heap values. IDA Pro allows us to trace those values at any particular program point. Since invariant generation techniques and static analysis techniques based on abstract interpretation struggle with the difficult problem of discovering loop invariant, we focus our dynamic analysis on loops since our static analyzer can easily deal with loop-free binary code. Dynamic analysis can often compute assertions that static analysis can not compute. Figure 3 illustrates this point. The figure describes the control flow of a function `Cntrl` with two arguments arg_0 and arg_4 of type integer. Both arguments arg_0 and arg_4 are decremented until arg_0 reaches the value 0. At that point, if the variables are equal then the function returns a 0, or a 1 otherwise.

We run the application in the IDA Pro debugger on few inputs and we obtain the following traces showing the values of arg_0 and arg_4 at various points of the program:

```
...
lea  eax,[ebp+arg_4]   ; Stack[]:arg_0: 00 00 00 00
...
mov  eax,[ebp+arg_0]   ; EAX=0
cmp  eax,[ebp+arg_4]   ; Stack[]:arg_0: 00 00 00 00
jnz  short loc_4831B   ; Stack[]:arg_4: 00 00 00 00
...
```

Imagine that this function is called with two values v_1 and v_2 at the beginning of a large application and depending on the result returned, a substantially different sub parts of the application's code is executed. All of the statical analysis approaches to intrusion detection known in the literature



```
Cntrl :
L1 :      push   ebp
L2 :      mov    ebp, esp
L3 :      sub    esp, 4
L4 :      jmp    short L_48304

L_48304 :
L1 :      cmp    [ebp + arg_0], 0
L2 :      jnz    short L_482FC

L_482FC :
L1 :      dec    [ebp + arg_0]
L2 :      lea    eax, [ebp + arg_4]
L3 :      dec    dword ptr [eax]

Block_1 :
L1 :      mov    eax, [ebp + arg_0]
L2 :      cmp    eax, [ebp + arg_4]
L3 :      jnz    short L_4831B

L_4831B :
L1 :      mov    [ebp + var_4], 1

Block_2 :
L1 :      mov    [ebp + var_4], 0
L2 :      jmp    short L_48322

L_48322 :
L1 :      mov    eax, [ebp + var_4]
L2 :      leave
L3 :      retn
```

Figure 3. Quasi-Static Analysis Example

will either ignore such a function because it does not refer to system calls, and therefore allow both sub parts to be executed, or will analyze such a function and will determine that the only invariant that can be extracted and the exit point of the function is `eax = 1 or eax = 1` no matter what arguments are passed to the function. Static analysis techniques based on abstract interpretation will generate the same invariant. This is mainly due to the presence of a loop who's body is the block labeled `L_482FC`. While an abstract interpreter will track the values of `[ebp+arg_0]`, that is, the value of the variable representing the first argument, and the value of the memory location who's address is in `[eax]`, that is, `[ebp+arg_0]`, the value of the variable representing the second argument, it will not converge and will generate the following invariant at the end of the loop `[ebp+arg_0] = 0 and [ebp+arg_4] =` $\perp$`]`. In other words, we know that at the exit point of the loop, `[ebp+arg_0] = 0` is true, which is the condition of exiting the loop, but any information about `[ebp+arg_4]` is lost. Using Daikon, we generate a set of candidate invariants. In particular, Daikon generates the following loop constraint indicating that the difference between the two arguments is constant:

$$\texttt{arg\_0\_482FC} - \texttt{arg\_4\_482FC} = \texttt{arg\_0} - \texttt{arg\_4}$$

It is easy to prove using Yices that such a candidate invariant is indeed an invariant. Using this constraint, we are able to summarize function `Cntrl` using the following predicates:

$$\texttt{arg\_0} = \texttt{arg\_4} \text{ implies } \texttt{eax} = 0$$
$$\texttt{arg\_0} \neq \texttt{arg\_4} \text{ implies } \texttt{eax} = 1$$

4.3 Using Yices Interface as an Abstract Interpreter and Assertion Checker

Our static analyzer is implemented using just three Yices commands that represents a simple interface to a very powerful decision procedure.

- `(assert+ fml)`: this commands adds the formula `fml` to the context. To each asserted formula, a unique id in the form of a positive integer is associated.
- `(check)` : this commands checks whether the conjunction of the already asserted formulas is satisfiable. That is, the conjunction does not contain a subset of inconsistent formulas.
- `(retract id)` : This command removes a formula with id `id` from the context.

When analyzing a block of instructions, we assert a formula for each instruction in the block. The formula is the semantics of the execution of the instructions.

4.4 Using Unsatisfiable Core to Propagate Invariants

When invoking the `check` command, Yices checks whether the already asserted formulas are satisfiable or not. The result of Yices can be `sat` indicating that the context is satisfiable and therefore the conjunction of the formulas in the context is an invariant of the program at the current program location. When the context is unsatisfiable, Yices returns with `unsat` $\texttt{id}_1$ $\texttt{id}_2$ $\texttt{id}_3$ ... $\texttt{id}_k$ indicating that the subset of formulas $\texttt{id}_1$ $\texttt{id}_2$ $\texttt{id}_3$ ... $\texttt{id}_k$ is inconsistent. Since the recently added formula reflects the semantics of the execution at the current control point, it remains in the context, and any formula in the subset that causes the inconsistency has to be removed.

5. Dealing With Obfuscation Techniques

Binary programs often exhibit various levels of obfuscation. Sometimes these obfuscation are intended to defeat static analysis. Some obfuscations can be the result of compiler optimization that produces an efficient code that is hard to analyze. In the following simple example, we show how our static analysis tool-bus based on Yices allows us to easily analyze certain classes of obfuscated programs. The example (Figure 4) describes a small program where a function `Main` calls the function `Max` that computes the max of 2 and 4. The arguments 2 and 4 are pushed onto the stack as well as the return address `L5`, and the program jumps to the function `Max`. Figure 5 shows the same example where the same

```
Main :                          Max :
L1 :    push  4                 L6 :   mov  eax, [esp + 4]
L2 :    push  2                 L7 :   mov  ebx, [esp + 8]
L3 :    push  offset L5         L8 :   cmp  eax, ebx
L4 :    jmp   Max               L9 :   jg   L11
L5 :    ret                     L10 :  mov  eax, ebx
                                L11 :  ret  8
```

Figure 4. Function call obfuscation using `push`/`jmp`

```
Main :                          Max :
L1 :    push  4                 L7 :   mov  eax, [esp + 4]
L2 :    push  2                 L8 :   mov  ebx, [esp + 8]
L3 :    push  offset L6         L9 :   cmp  eax, ebx
L4 :    push  offset Max        L10 :  jg   L12
L5 :    ret                     L11 :  mov  eax, ebx
L6 :    ret                     L12 :  ret  8
```

Figure 5. Function call obfuscation using `push`/`ret`

functionality is achieved using the `push` and `ret` instructions. Figure 6 shows the same example where the same functionality is achieved using the `push` and `ret` instructions. Translating the semantics of the three programs into

```
Main :                Max :
L1 :    push  4       L5 :   mov  eax, [esp + 4]
L2 :    push  2       L6 :   mov  ebx, [esp + 8]
L3 :    call  Max     L7 :   cmp  eax, ebx
L4 :    ret           L8 :   jg   L10
                      L9 :   mov  eax, ebx
                      L10 :  pop  ebx
                      L11 :  add  esp, 8
                      L12 :  jmp  ebx
```

Figure 6. Function call obfuscation using `pop` to `return`

Yices, leads to equivalent states at the end of the execution of the three programs. If we modify the third program (Figure 7) by making the call to `Max` dependent of the return value of the function `Ctrl` (Figure 3), we can prove that function Max will never be called.

```
Main :                      Max :
L1 :    push  3             L5 :   mov  eax, [esp + 4]
L2 :    push  5             L6 :   mov  ebx, [esp + 8]
L3 :    call  Cntrl         L7 :   cmp  eax, ebx
L13 :   test  eax, eax      L8 :   jg   L10
L14 :   jnz   L18           L9 :   mov  eax, ebx
L15 :   push  4             L10 :  pop  ebx
L16 :   push  2             L11 :  add  esp, 8
L17 :   call  Max           L12 :  jmp  ebx
L18 :   ret
```

Figure 7. Function call obfuscation using `pop` to `return`

6. Conclusion

Different static analysis tools operate on different abstract representation of a program. They are engines for generating different invariants of the same program. Abstract state graphs computed by predicate abstraction can be viewed as an intermediate structure that allows each tool to share its results and to take advantage of the analysis performed by other tools. In fact, the analysis of each tool can be viewed as a refinement of the abstract state graph. We have shown how dynamic analysis and decision procedures can be combined with static analysis to build more precise models of binary programs. We believe that a tool-bus architecture operating on abstract state graphs is an efficient and general approach to combining and extending static analysis tools in order to prove more complex properties of software while being scalable and computationally efficient. Furthermore, the nondeterminism in the abstract state graph indicate what part of the system requires more analysis.

Acknowledgments

The ideas described in this paper were inspired by many discussions with my colleagues Bruno Dutertre, Sam Owre, John Rushby, Natarajan Shankar and Ashish Tiwari.

References

[1] IDA Pro Dissasember . http://www.datarescue.com/ida.htm.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Compiler Construction (LNCS 2985)*, pages 5–23. Springer Verlag, Apr. 2004.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, Jan. 2002.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, Jan. 1977.

[5] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). In *The 18th Computer-Aided Verification*

Conference (CAV'06), Seattle, June 2006. Lecture Notes in Computer Science.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[7] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer Verlag.

[8] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM Press.

[9] W. McCloskey, S. Gulwani, and A. Tiwari. Lifting abstract interpreters to quantified abstract domains. In *Proc. Principles of Programming Languages, POPL*, 2008. To appear.

[10] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607, pages 748–752. Springer Verlag, June 1992.

[11] H. Saïdi. Guarded models for intrusion detection. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 85–94, New York, NY, USA, 2007. ACM Press.

[12] A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNAI*, pages 147–166. Springer, 2007.

[13] P. Vales, J. Butler, D. Rager, C. Stack, and C. Telfer. Gaps in static analysis tools coverage. In *OMGs First Software Assurance Workshop: Working Together for Confidence*, Fairfax, VA, mar 2007.