REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
16-11-2015	Final Report	15-Aug-2009 - 14-Aug-2012

4. TITLE AND SUBTITLE
Final Report: [5.5 Information and Software Assurance:] Automated Vulnerability Detection in Executables

5a. CONTRACT NUMBER
W911NF-09-1-0413

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS
Thomas Reps

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
University of Wisconsin - Madison 1.00
Suite 6401
21 N Park St
Madison, WI 53715 -1218

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
56037-CS.15

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT
The primary goal of this project is to create tools to automate detection of security vulnerabilities in machine-code executables. The aim is to generate a list of definite vulnerabilities and information about what is required to exploit them. The focus of the work is on creating a machine-code version of the DASH model checker, which is a hybrid concrete/symbolic program-exploration algorithm for performing goal-directed white-box ``fuzzing'' of source-code programs.

15. SUBJECT TERMS
machine-code analysis, detection of security vulnerabilities

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas Reps
UU	UU	UU	UU		19b. TELEPHONE NUMBER 608-262-2091

Report Title

Final Report: [5.5 Information and Software Assurance:] Automated Vulnerability Detection in Executables

ABSTRACT

The primary goal of this project is to create tools to automate detection of security vulnerabilities in machine-code executables. The aim is to generate a list of definite vulnerabilities and information about what is required to exploit them. The focus of the work is on creating a machine-code version of the DASH model checker, which is a hybrid concrete/symbolic program-exploration algorithm for performing goal-directed white-box ``fuzzing'' of source-code programs.

Enter List of papers submitted or published that acknowledge ARO support from the start of the project to the date of this printing. List the papers, including journal references, in the following categories:

(a) Papers published in peer-reviewed journals (N/A for none)

Received		Paper
11/16/2015	1.00	Junghee Lim, Akash Lal, Thomas Reps. Symbolic analysis via semantic reinterpretation, International Journal on Software Tools for Technology Transfer, (05 2011): 61. doi: 10.1007/s10009-010-0158-6
11/16/2015	2.00	Peter Lammich, Nicholas Kidd, Tayssir Touili, Thomas Reps. A decision procedure for detecting atomicity violations for communicating processes with locks, International Journal on Software Tools for Technology Transfer, (06 2011): 0. doi: 10.1007/s10009-010-0159-5
11/16/2015	3.00	Nicholas Kidd, Thomas Reps, Julian Dolby, Mandana Vaziri. Finding concurrency-related bugs using random isolation, International Journal on Software Tools for Technology Transfer, (06 2011): 495. doi: 10.1007/s10009-011-0197-7
11/16/2015	5.00	Matt Elder, Denis Gopan, Thomas Reps. View-Augmented Abstractions, Electronic Notes in Theoretical Computer Science, (01 2010): 43. doi: 10.1016/j.entcs.2010.09.005
TOTAL:	**4**	

Number of Papers published in peer-reviewed journals:

(b) Papers published in non-peer-reviewed journals (N/A for none)

Received		Paper
11/16/2015	10.00	Anupam Datta, Somesh Jha, Ninghui Li, David Melski, Thomas Reps. Analysis Techniques for Information Security, Synthesis Lectures on Information Security, Privacy, and Trust, (04 2010): 0. doi: 10.2200/S00260ED1V01Y201003SPT002
TOTAL:	**1**	

Number of Papers published in non peer-reviewed journals:

(c) Presentations

Number of Presentations: 0.00

Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received		Paper
11/16/2015	6.00	Bill McCloskey, Thomas Reps, Mooly Sagiv. Statically Inferring Complex Heap, Array, and Numeric Invariants, Static Analysis Symposium (SAS). 14-SEP-10, . : ,
11/16/2015	12.00	Aditya V. Thakur, Matt Elder, Thomas W. Reps. Bilateral Algorithms for Symbolic Abstraction, Static Analysis Symposium (SAS). 11-SEP-12, . : ,
TOTAL:	**2**	

Number of Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received		Paper
11/16/2015	4.00	William R. Harris, Somesh Jha, Thomas Reps. DIFC programs by automatic instrumentation, 7th ACM Conference on Computer and Communications Security (CCS). 03-OCT-10, Chicago, Illinois, USA. : ,
11/16/2015	8.00	Evan Driscoll, Amanda Burton, Thomas Reps. Checking conformance of a producer and a consumer, Found. of Software Engineering (FSE). 04-SEP-11, Szeged, Hungary. : ,
11/16/2015	9.00	Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors, Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS). 04-MAR-11, Newport Beach, California, USA. : ,
11/16/2015	11.00	Aditya Thakur, Thomas Reps. A Generalization of Stålmarck's Method, Static Analysis Symposium (SAS). 11-SEP-12, . : ,
11/16/2015	13.00	Evan Driscoll, Aditya Thakur, Thomas Reps. OpenNWA: A Nested-Word Automaton Library, Computer-Aided Verification (CAV). 07-JUL-12, . : ,
11/16/2015	14.00	Aditya Thakur, Thomas Reps. A method for symbolic computation of abstract operations, Computer-Aided Verification (CAV). 07-JUL-12, . : ,
TOTAL:	**6**	

Number of Peer-Reviewed Conference Proceeding publications (other than abstracts):

(d) Manuscripts

Received	Paper
TOTAL:	

Number of Manuscripts:

Books

Received Book

TOTAL:

Received Book Chapter

TOTAL:

Patents Submitted

Patents Awarded

Awards

2010 ACM SIGSOFT Retrospective Impact Paper Award (for Reps, T. and Teitelbaum, T., "The Synthesizer Generator," 1984)

2011 ACM SIGSOFT Retrospective Impact Paper Award (for Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," 1994)

In 2013, Reps was elected a Foreign member of Academia Europaea

Junghee Lim received the UW Computer Sciences Department's Outstanding Graduate Student Research Award for 2010-2011

Aditya Thakur (who finished his dissertation in 2014) was a co-recipient of the UW Computer Sciences Department's Outstanding Graduate Student Research Award for 2013-2014.

Graduate Students

NAME	PERCENT_SUPPORTED	Discipline
Junghee Lim	0.00	
Aditya Thakur	0.25	
Tycho Anderson	0.05	
Tushar Sharma	0.14	
Prathmesh Prabhu	0.25	
FTE Equivalent:	**0.69**	
Total Number:	**5**	

Names of Post Doctorates

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Faculty Supported

NAME	PERCENT_SUPPORTED	National Academy Member
Thomas Reps	0.14	
FTE Equivalent:	**0.14**	
Total Number:	**1**	

Names of Under Graduate students supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Student Metrics

This section only applies to graduating undergraduates supported by this agreement in this reporting period

The number of undergraduates funded by this agreement who graduated during this period: 0.00

The number of undergraduates funded by this agreement who graduated during this period with a degree in science, mathematics, engineering, or technology fields:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and will continue to pursue a graduate or Ph.D. degree in science, mathematics, engineering, or technology fields:..... 0.00

Number of graduating undergraduates who achieved a 3.5 GPA to 4.0 (4.0 max scale):..... 0.00

Number of graduating undergraduates funded by a DoD funded Center of Excellence grant for Education, Research and Engineering:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and intend to work for the Department of Defense 0.00

The number of undergraduates funded by your agreement who graduated during this period and will receive scholarships or fellowships for further studies in science, mathematics, engineering or technology fields: 0.00

Names of Personnel receiving masters degrees

NAME	
Prathmesh Prabhu	
Tycho Anderson	
Total Number:	**2**

Names of personnel receiving PHDs

NAME	
Junghee Lim (2011)	
Evan Driscoll (2013)	
Aditya Thakur (2014)	
Total Number:	**3**

Names of other research staff

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Sub Contractors (DD882)

Inventions (DD882)

Scientific Progress

See Attachment

Technology Transfer

Starting in Fall 2007, Reps's group began to transition to GrammaTech the TSL technology, and somewhat later the TSL-developed analysis components for dynamic-analysis, static-analysis, and symbolic-analysis (including versions of the analysis components that are incorporated in McVeto). At GrammaTech, this technology has played an important role in 6 SBIR/STTR Phase I projects, 11 SBIR/STTR Phase II projects, as well as four other GrammaTech contracts, including a $13M IARPA project funded under STONESOUP (recently approved to enter Phase 2), as well as a multi-million dollar contract under DARPA's RAPID program.

The logic-based analysis components generated using TSL (which were originally developed for use in McVeto) have been embraced at GrammaTech for use in CodeSonar/x86, a machine-code version of GrammaTech's CodeSonar bug-finding product for source code (C and C++). GrammaTech began the development of CodeSonar/x86 as part of a GrammaTech/SRI Army STTR Phase II project ("Semantics-Aware Malware Detection", W911NF-07-C-0003). Using TSL, CodeSonar/C has been retargeted to run on programs written in the Intel IA32 instruction set (a.k.a. x86). CodeSonar/x86 has not yet been released as a product; however, versions of CodeSonar/x86 have been delivered to early adopters in the intelligence community (e.g., AndyWhite in NSA I7, and others), and it has been put into use. CodeSonar/x86 was also delivered to the NSA SPO for Supply Chain Risk Management. CodeSonar/x86 also played a role in an MDA SBIR Phase I project on insider threat. The incorporation of techniques inspired by McVeto into CodeSonar/x86 is the subject of a GrammaTech/UW Navy STTR Phase II contract that began in September 2011.

Automated Vulnerability Detection in Executables
W911NF-09-1-0413
Professor Thomas Reps, University of Wisconsin

Objective

Our goal is to develop *methods to automate detection of security vulnerabilities in machine-code executables*. The aim is to boost the capabilities of human analysts by providing them with a method that identifies vulnerabilities and information about what is required to exploit them. Our approach is to leverage recent advances in (a) state-space-exploration and (b) automated-deduction techniques to develop techniques that (i) automate the task of finding security vulnerabilities, and (ii) are capable of finding vulnerabilities that analysts would otherwise not find, or find only with great effort.

Approach

As initially conceived, the goal of the project was to create a machine-code analog of the DASH model checker—a tool that we dubbed "McVeto" (for Machine-Code Verification Tool). While model checkers such as SLAM, BLAST, and DASH have each made significant contributions in the field of verification/flaw-detection, their use has been restricted to programs for which source code is available. The project addresses the challenges of creating a model checker that is (i) capable of verifying properties of machine-code programs (or finding vulnerabilities in them), and (ii) can be retargeted easily to different instruction sets.

DASH is a hybrid concrete/symbolic program-exploration algorithm developed at Microsoft Research, Bangalore for performing goal-directed white-box "fuzzing" of C source-code programs. DASH has the interesting property that by using symbolic reasoning in combination with testing runs, it can sometimes verify that a (bad) goal state cannot be reached (refuting Dijkstra's famous quote that "testing can be used to show the presence of bugs, but never to show their absence"). Alternatively, DASH might find an initial state that definitely causes the bad state to be reached. The third possibility is that DASH may fail to terminate.

For exploit discovery, the goal state for McVeto would be one that allows the attacker to achieve some malicious outcome. The DASH algorithm attempts to establish that the goal state is unreachable; thus, *any counterexample reported shows that a vulnerability definitely exists, and provides an input for exploiting it*.

The basic DASH algorithm maintains one data structure that consists of concrete traces that have been explored, together with a second data structure that represents an abstraction of the entire state space. At appropriate moments, the pre-image operation is used to create new predicates that refine regions of the abstracted state space. In some cases, DASH can establish that there is no path from program entry to the "goal" state with which it has been furnished; when this happens DASH has verified that the "goal" is unreachable.

McVeto was originally called McDash. We changed the name so that others would appreciate better the different goals of McVeto vis a vis the original DASH tool (i.e., machine-code analysis versus source-code analysis), as well as the fact that McVeto incorporates a number of different analysis techniques beyond what is used in DASH.

Scientific Opportunities and Barriers

The challenge in creating McVeto is to take a promising approach that has been developed for source-code programs (where one can work with the assumption that you can build a control-flow graph as an initial abstraction of the program; you know what the variables in the program are, as well as their types; etc.) and to find a way to apply it to machine code (a context in which, in general, one cannot build an accurate control flow graph, and where one does not know what the variables are, let alone their types).

On the other hand, working with machine code also presents a scientific opportunity insofar as machine-code is an artifact that, compared with source code, is closer to what is actually executed on a machine. That is, the transformation from source code to machine code can introduce subtle but important differences between what a programmer intended and what is actually executed by the processor. Some of the reasons why analyses based on source code can provide the wrong level of detail include

- Many security exploits depend on platform-specific details that exist because of features and idiosyncrasies of compilers and optimizers. These include memory-layout details (such as the positions—i.e., offsets—of variables in the runtime stack's activation records and the padding between structure fields), register usage, execution order (e.g., of actual parameters at a call), optimizations performed, and artifacts of compiler bugs.
- Analyses based on source code typically make (unchecked) assumptions, e.g., that the program is ANSI C compliant. These assumptions often mean that an analysis does not account for behaviors that are allowed by the compiler and that can lead to bugs or security vulnerabilities (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.)
- Programs are sometimes modified subsequent to compilation, e.g., to perform optimizations or insert instrumentation code. They may also be modified to insert malicious code. Such modifications are not visible to tools that analyze source code.

In short, even when source code is available, a substantial amount of information is hidden from source-code-analysis tools, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools.

Although having to perform static analysis on machine code represents a daunting challenge, there is also a possible silver lining: by analyzing an artifact that is closer to what is actually executed, a static-analysis tool may be able to obtain a more accurate picture of a program's properties. The reason is that—to varying degrees—the semantic definition of every programming language leaves

certain details unspecified. Consequently, for a source-code analyzer to be sound, it must account for all possible implementations, whereas a machine-code analyzer only has to deal with one possible implementation—namely, the one for the code sequence chosen by the compiler.

Moreover, the DASH approach of using testing runs offers another opportunity vis à vis the challenges of machine-code analysis. In particular, machine code has two properties not found in source code: (i) it can be self-modifying, and (ii) the byte values of machine code can have multiple "readings" when decoded starting from different addresses (i.e., a given sequence of instructions has a second meaning when decoded out-of-registration with the original decoding of the instruction bytes). Such issues present challenges for applying standard source-code-analysis techniques, including such simple concepts as building a control-flow graph (CFG) as an initial abstraction of the program. Instead, as we have shown in McVeto, the testing runs can be used to learn a data structure that serves as a suitable substitute for a CFG when working with machine code.

Significance

The significance of the project is its potential to create better technology for identifying vulnerabilities in software, and understanding their potential for being exploited. The work on McVeto could provide techniques for creating tools that find more, and better-quality, vulnerabilities/exploits in ways that are faster and cheaper. Such tools could have a lower reliance on subject-matter experts, leverage extensive hardware automation, have a lower reliance on undirected random search ("fuzzing"), focus more on logical boundary conditions, and thus have greater precision and recall. These are significant advantages from both the defensive and offensive perspectives.

For exploit discovery, the appealing aspect of McVeto is that its exploration is performed with the intention of refuting the assertion that a set of goal states (e.g., error states) is reachable. For exploit discovery, the goal states would be those that allow the attacker to achieve some malicious outcome. McVeto attempts to establish that no vulnerability exists (and hence can be used for verification), but *any counterexample reported represents an actual vulnerability*.

A drawback of most program-analysis tools is that they spread their effort homogeneously across the program, and hence expend considerable effort establishing properties in regions of the program that may not be of interest to an analyst. Thus, another goal is to use a McVeto-like mixture of concrete execution and abstract execution to create a tool that provides a "fisheye" view of machine code. That is, if one knows that region R of the code is an area of particular interest, one wants more details about R—greater "magnification" of R—and fewer details about everything outside of R. In addition to less detail, the tool should minimize the effort

that it spends analyzing parts of the program outside of R, and invest extra effort analyzing R itself. The project will attempt to create a tool with such properties by biasing McVeto's search to consider paths that pass through R on the way to the goal site.

Unlike previous model checkers, McVeto does not require the usual preprocessing steps of (a) building control-flow graphs, and (b) performing points-to analysis (or alias analysis); nor does McVeto require type information to be supplied. McVeto is also able to check properties of self-modifying code. Moreover, by using language-independent meta-tools that generate the implementations of the required analysis components from descriptions of an instruction set's syntax and semantics, we will be able to create versions of McVeto for different instruction sets automatically.

Accomplishments (2011-12)

- Continued work on the use of interpolation and similar approaches in McVeto
  - Continued collaboration with Ken McMillan (Microsoft Research) to address several problems that arose when using Ken's Foci interpolant generator on machine code
  - Developed an alternative to interpolation, based on unsat-cores. The use of unsat-cores has benefits similar to interpolation: smaller formulas for the abstraction-refinement predicates used by McVeto, and they also provide a degree of generalization due to the fact that unsat-cores discard conjuncts of the original formula—although there is not the same shared-vocabulary constraint as one obtains from interpolants.
- Developed an algorithm that can use either of our two implementations of affine-relation analysis to create loop and procedure summaries. All summaries are computed with a single call to our analyzer (i.e., one "poststar" query). The analyzer also augments the semantics of the program with artificial trip-count variables (a la Saxena et al. ISSTA 2009); the loops considered are the program's Bourdoncle components.
- My student Aditya Thakur and I pursued a new insight on how to perform program verification, create abstractions, and find invariants. Our work resurrects an old theorem-proving technique, called Stålmarck's method, and applies its ideas in new ways in the context of program analysis. This approach holds great promise for a multitude of fundamental tasks in program analysis. Moreover, our methods are "dual-use": in addition to their use for program analysis, they hold the promise of helping with the creation of improved logic solvers (which are playing key roles in many program-analysis tools these days).
- One of the techniques used in McVeto is to identify *candidate* program invariants, which are then added to McVeto's over-approximating program abstraction in a way that preserves the over-approximation. If the McVeto proof process can establish that the candidate invariants are indeed invariants, this can speed up the overall proof process. My student Prathmesh Prabhu and I have developed an algorithm that has the promise to speed up the ability to identify candidate invariants in McVeto – and more broadly, to provide a method for performing improved dataflow analysis in *any* kind of program-analysis tool. Our

approach is based on work by Esparza et al. at the Technical Univ. of Munich to apply a variant of Newton's method (in a non-numerical context – namely dataflow analysis) to solve the equation systems that arise in a dataflow-analysis problem. However, the work of Esparza et al. has focused on the case where the analog of multiplication is a *commutative* operator, whereas in the dataflow-analysis context the analog of multiplication is *non-commutative*. Our version of the Newton-inspired algorithm is being incorporated into WALi, which is a generic framework for performing dataflow analysis developed in my group.

Collaborations and Leveraged Funding

The project is closely coupled with several other projects that Reps is carrying out, supported by AFOSR, NSF, and ONR. The AFOSR and ARL projects are aimed at exploiting the capabilities of so-called "directed proof generation", which is a method to sniff out bugs and security vulnerabilities in a focused manner. It performs a goal-directed search to see if a given target of interest can be reached; if the target is unreachable, the outcome serves as a proof that the program is correct. The ARL project concentrates on building a version of directed proof generation for machine code. The AFOSR project concerns how to combine a search of machine code with a similar search starting from the source code.

Reps's group is also collaborating on several projects with GrammaTech, Inc. GrammaTech provides a technology-transfer pipeline for the basic-research results on machine-code analysis obtained at Wisconsin to be transitioned into the hands of working analysts in a relatively short time. (See the section on "Technology Transfer" below.)

In all of these projects, the analysis tools are built using a language-independent machine-code-analysis toolkit, originally developed at Wisconsin with IARPA support, and now with NSF support. The toolkit, called TSL, is a unique *analyzer-creation tool* for *generating* dynamic-analysis, static- analysis, and symbolic-analysis components for programs written in different programming languages ("subject languages"). The TSL system consists of (i) a language-specification language, also called TSL (for "Transformer Specification Language"), for describing the semantics of a subject language, along with (ii) a processing system for automatically generating multiple analysis components from the TSL specification of the subject language, and (iii) a run-time system to support the analysis of programs written in the subject language. TSL allows analyzers with multiple analysis components to be created for multiple languages with greatly reduced effort. In particular, TSL provides a way to leverage the work performed to create tools for one language so that it applies to many languages. Because of the flexibility that the TSL system offers, one can explore the design spaces of a large range of problems with a relatively small group, whereas a conventional approach would require much more in the way of resources to cover the same territory.

Conclusions

- McVeto definitely works on small examples
- McVeto shows promise on security-sensitive benchmarks
    - On 106 stripped executables generated from the Verisec suite that the current version of McVeto can handle, we obtained the following results:
        - In 80 cases, McVeto identified a buffer overrun, and supplied a program input that provoked it
        - In 26 cases, McVeto proved that *no* input to the benchmark can provoke an exploitable buffer overrun.
- Interpolation shows promise, although so far only on small examples.
- Stålmarck's method holds great promise for building more powerful tools for program analysis. We believe that this work has the potential to revolutionize the way program-analysis problems are thought about, and how program-analysis tools are constructed.
- The jury is still out on whether Newton's method is faster than conventional methods when solving real program-analysis problems.

Technology Transfer

Starting in Fall 2007, Reps's group began to transition to GrammaTech the TSL technology, and somewhat later the TSL-developed analysis components for dynamic-analysis, static-analysis, and symbolic-analysis (including versions of the analysis components that are incorporated in McVeto). At GrammaTech, this technology has played an important role in 6 SBIR/STTR Phase I projects, 11 SBIR/STTR Phase II projects, as well as four other GrammaTech contracts, including a $13M IARPA project funded under STONESOUP (recently approved to enter Phase 2), as well as a multi-million dollar contract under DARPA's RAPID program.

The logic-based analysis components generated using TSL (which were originally developed for use in McVeto) have been embraced at GrammaTech for use in CodeSonar/x86, a machine-code version of GrammaTech's CodeSonar bug-finding product for source code (C and C++). GrammaTech began the development of CodeSonar/x86 as part of a GrammaTech/SRI Army STTR Phase II project ("Semantics-Aware Malware Detection", W911NF-07-C-0003). Using TSL, CodeSonar/C has been retargeted to run on programs written in the Intel IA32 instruction set (a.k.a. x86). CodeSonar/x86 has not yet been released as a product; however, versions of CodeSonar/x86 have been delivered to early adopters in the intelligence community (e.g., Andy White in NSA I7, and others), and it has been put into use. CodeSonar/x86 was also delivered to the NSA SPO for Supply Chain Risk Management. CodeSonar/x86 also played a role in an MDA SBIR Phase I project on insider threat. The incorporation of techniques inspired by McVeto into CodeSonar/x86 is the subject of a GrammaTech/UW Navy STTR Phase II contract that began in September 2011.