Theory and Practice of Mechanized Software Analysis

J. Aaron Pendergrass, Susan C. Lee, and C. Durward McDonell

As software systems become ever more vital to all aspects of daily life, the risks posed by defects in critical software become increasingly dire. Traditional software engineering techniques focus heavily on manual analysis and testing to discover and repair defects. Although this approach is valuable, modern tools for the mechanized or automated detection of defects have proven themselves capable of alleviating much of the tedium associated with manual processes while providing greater assurance in their coverage. In this article, we describe the strengths and weaknesses of the most common approaches to the automated detection of software defects: formal methods and source code verification. We then describe our experience applying both free and commercial tools based on these techniques in the Software Analysis Research and Applications Laboratory (SARA Lab), a new effort at APL to enhance the state of the art in software analysis while applying best-of-breed tools for defect detection to APL software projects. APL software developers can avail themselves of this research by by e-mailing +SARALab.

INTRODUCTION

Formal verification offers great promise for establishing confidence that software implements its intended functionality. Although effective, Formal Methods (FM) techniques generally scale poorly, are difficult for nonexperts to learn, and impose a high up-front cost on development efforts. To expand their use, FM require additional research and development to improve their usability. Other techniques in source-code verification that trade off expressiveness or soundness for developer ease of use also offer great promise for enhancing the security posture of software and providing a low-cost path toward adoption of more rigorous verification techniques. This article describes the Software Analysis Research and Applications Laboratory (SARA Lab), an effort to further the application of software analysis techniques to APL-developed technologies and to leverage APL's software development capabilities to advance the state of the art in software analysis.

FORMAL METHODS

FM fall into one of two categories: model checking or deductive verification. These categories differ in terms of the range of different systems that a formal method can describe as well as the reasoning strategy.

To be analyzed by a model checker, a system must be amenable to modeling in a fairly limited way, often as some sort of state machine. The sound reasoning strategy for model checkers is a brute-force exploration of the state space of the system model, proving that it will never enter some user-defined undesirable state. Moreover, a model checker can prove that the system *can* enter some user-defined desirable state; this is called a liveness property. Model checkers can be automated, but they are limited by the size of the system (i.e., by the number of states to be explored within the time and computing power available to the analyst).

Deductive verification proves properties about a program by first describing the valid input states of the program as a logical predicate and then applying inference rules corresponding to each command in the program to transform this precondition into a similar predicate describing the program's final state. This process is similar to traditional mathematical proofs, such as the two-column proofs familiar to many people from high school geometry. Most systems for applying deductive verification to software require a great deal of manual effort to select which inference rules to apply at each step to guide the proof toward the desired goal. "Proof assistants" (i.e., tools that can perform part of the work for the analyst) exist for some reasoning systems; however, proofs cannot be fully automated in all cases and require human intervention and guidance to verify that the property we desire is true (or not).

STATIC ANALYSIS TECHNIQUES

Numerous commercial, open-source, and research tools exist for automatically finding defects in source code. Probably the best known of these is `lint`, a tool developed in tandem with the original C compiler intended to flag suspicious constructs.[1] The choice to separate this functionality from the compiler, largely motivated by efficiency concerns for the compiler, was probably one of the watershed moments that led to the widespread lack of adoption of such automated program analysis tools in the field. Now, as compiler efficiency concerns have taken a backseat to program correctness and security, static checkers have become increasingly popular. The SARA Lab aims to increase the appeal of such static checkers by improving the quality and depths of their analyses.

Unlike more heavyweight FM proof techniques, static analysis tools are designed to be automated, fast, and usable by a broad population of programmers. This makes static analysis an important part of enforcing good coding practices, but it also means that static analysis tools are generally unable to provide the same insight on specific correctness requirements of a project that FM might provide. For example, FM techniques can be used to prove that a routine intended to sort a list actually returns its input in sorted order. Because static analysis tools have no notion of what algorithm a routine is supposed to implement, they focus on identifying common classes of defects such as NULL pointer dereferences, memory leaks, and buffer overflows that, if present, are always undesirable. (A NULL pointer is a reference to an undefined cell of computer memory, typically represented by the number 0. A dereference of a NULL pointer produces undefined program behavior. A memory leak is a behavior of a program that allocates memory to hold some temporary data but fails to properly deallocate this memory when it is no longer in use. A buffer overflow occurs when a program attempts to write data to a region of memory that is not large enough to hold the data being written.) To cope with large code bases and the often-difficult-to-understand nature of complex software, static analysis tools usually make additional compromises to ensure speedy analysis. No complex software analysis can provide a complete analysis—one that will label all valid programs as good. To limit false-alarm rates, most static analysis tools will also sacrifice soundness, causing them to report some invalid programs as good. Despite these limitations, static analysis tools are capable of consistently detecting a wide range of suspicious or invalid programs.

There are several common techniques implemented by most static analysis tools. Each technique offers a different trade-off of among results, precision, false-positive rates, and computational requirements. Most tools include a combination of techniques to provide a reasonable balance between each one's strengths. The major classes of static analysis are as follows:

- Syntactic pattern matching
- Type systems
- Data-flow analysis
- Abstract interpretation

Syntactic Pattern Matching

Syntactic pattern matching is the fastest and easiest technique for static analysis, but it provides little confidence in program correctness and can result in a high number of false alarms. A checker based on syntactic pattern matching works by defining a set of program constructs that are potentially dangerous or invalid and then searching the input program's abstract syntax tree for instances of any of these constructs. A common example for C programs is a pattern to prevent the use

of an assignment expression, `<lhs> = <expr>`, as the condition of an if-then-else block. This pattern represents a common error in C programs, illustrated by Example 1, in which a programmer intends to use the comparison operator "==" to determine whether the two sides of the expression are equal, rather than using the assignment operator "=" to assign the value of the right-hand side to the variable or memory location given by the left-hand side. However, this pattern does represent perfectly valid C code and may be used intentionally by a programmer, as shown in Example 2, to assign a variable and branch based on whether the new value is 0.

```
if (n = 3) {
    ...
} else {
    /* will never get here */
}
```

Example 1. Mistaken use of `<lhs> = <expr>` construct.

```
if (p = malloc(...)) {
    /* do this if p is not NULL, i.e., if malloc succeeded */
} else {
    /* do this if p is NULL, i.e., if malloc failed */
}
```

Example 2. Intentional use of `<lhs> = <expr>` in an if–else construct.

One of the most common uses for syntactic pattern matching is to automatically enforce coding style guidelines across an entire code base to improve the consistency and readability of code. Some syntactic pattern matching systems are able to automatically transform some invalid constructs to equivalent acceptable forms based on rewrite rules that substitute one pattern for another.

```
<high> FILE *confidential;
<low> FILE *public;
char buf[1024];
...
confidential = fopen("/important/data.txt", "r");
public = fopen("/usr/share/everyone", "w");
...
fread(buf, 1024, 1, confidential);
fwrite(buf, 1024, 1, public);
...
```

Example 3. With additional annotations, a type system can catch security-related information-flow errors such as this.

Type Systems

Type systems are a core part of programming languages and are familiar to most, if not all, programmers.[2] However, the use of type systems to enforce program correctness properties may not be immediately apparent to many. A type system assigns a label to each variable in a program and defines rules for how these labels can be combined and how they are propagated by the primitives of the programming language. The types `int` and `float` from the C language are common examples of type labels representing fixed-size integers (e.g., 32-bit integers) and floating point numbers, respectively. The C type system defines the result of built-in operators with respect to these labels; for example, adding an `int` to a `float` results in a new `float` value. Other rules in the type system forbid certain operators from being applied to variables with types that would not make sense; for example, it is illegal to attempt to treat a `float` as the address of a memory cell by applying the unary "*" (dereference) operator.

Type systems have application far beyond these simple examples. The same algorithms that are used to determine that attempts to dereference a `float` are illegal can be used to prevent data labeled as confidential from being written to an output channel labeled as public, as in Example 3. In addition to their application to information-flow properties such as this, type systems

have been applied to check for a wide range of program errors, including units of measurement being properly converted,[3] format string vulnerabilities,[4] and race conditions in concurrent software.[5] Advanced type systems such as the dependent types featured in the Coq proof assistant[6] can even be used to prove arbitrary program correctness properties.

One of the major technologies enabling the use of type systems as an automated static analysis tool is type inference: the ability to algorithmically determine the correct types to give to a program's variables. Type inference frees the programmer from having to explicitly annotate all of the variables in a program with additional, often complicated, type labels. Instead, the type inference algorithm uses the operations performed on each variable to constrain the set of type labels that make sense. If the constraints are satisfiable, then each program variable can be assigned a type and the program passes analysis; if the constraints are unsatisfiable, then some variables may not have a corresponding type and the program fails the analysis. This allows the programmer to specify only a small number of type labels, such as the sensitivity level of input and output channels, and the analysis can then determine whether the program meets a specified property.

Like all static analysis techniques, type systems may reject a program that is actually free of errors. The most common reason for this is that type systems are traditionally "flow-insensitive," meaning that once a variable is given a type label, that type label persists for the life of the variable. In Example 4, the variable "buf" is first assigned confidential data but is then updated with public data before being written to the public output channel. Although this example does not contain an actual information-flow violation, a flow-insensitive type system would label the variable "buf" as holding confidential data and thus would reject the program.

```
<high> FILE *confidential;
<low> FILE *public, *public2;
char buf[1024];
...
confidential = fopen("/important/data.txt", "r");
public = fopen("/usr/share/everyone", "w");
public2 = fopen("/usr/share/allusers", "r");
...
fread(buf, 1024, 1, confidential);
[do something with buf]
fread(buf, 1024, 1, public2);
fwrite(buf, 1024, 1, public);
...
```

Example 4. Correct information flow that can confuse many type systems.

Data Flow Analysis

Data flow analysis overcomes the challenges of type systems related to flow insensitivity by taking into account how the data referenced by a variable may change throughout a program.[7] At each source-code location, data flow analysis records a set of facts about all variables currently in scope. These facts describe some property of the variables in the program at each source-code location; the exact kinds of facts recorded are unique to the specific analysis being performed. One common example used is a "reaching definitions" analysis in which the fact set at each line of code indicates the location of the most recent assignment of each program variable. Other data flow-based analyses often used by optimizing compilers are summarized in Table 1. In addition to the set of facts to be tracked, the analysis defines a "kills" set and a "gens" set for each construct of the programming language. The "kills" set describes the set of facts that are invalidated by execution of the construct, and the "gens" set describes the set of facts that are generated by the construct. To analyze a program, the analysis tool begins with an initial set of facts and updates it according to the "kills" set and "gens" set for each statement of the program in sequence. Depending on the analysis being performed, the program statements may be processed either forward or backward, and the initial fact set may assume that nothing is true for all variables or that all variables share some property.

All of these data flow analyses in Table 1 are used by optimizing compilers to eliminate redundant computation, reorder statements, or temporarily reuse variables. However, these same analyses can be used to verify program correctness (e.g., by identifying that a pointer being dereferenced was last assigned a potentially NULL value). A similar analysis can be used to solve the information-flow problem of Example 4 by tracking the information-flow label of the reaching definition of "buf." At the time "buf" is output to the public channel, its reaching definition is associated with public data, so the program will pass the analysis.

Table 1. Four common examples of data flow analysis used in optimizing compilers

Analysis	Facts Tracked	Direction	Common Application	Example
Reaching definitions	Location of most recent assignment to each variable	Forward	Reordering of code to improve	(1) $x = y + 1$; (2) $x = x + 1$; (3) $w = z + 1$; Statements (3) and (2) may be reordered because the reaching definition of z is before statement (1). This may improve pipelining behavior.
Live variables	Variables whose values will be used later in the code	Backward	Elimination of unused assignments, reuse of memory to represent different variables	(1) $x = 3$; (2) exit; Statement (1) may be eliminated because the value of x is never used.
Available expressions	Arithmetic expressions that have been recently computed	Forward	Elimination of redundant arithmetic expressions	(1) $x = 12*7$; (2) $y = 12*7$; The computation of 12*7 in statement (2) is redundant and can be eliminated.

Very busy expressions	Arithmetic expressions that will be computed later in the code on any paths through the program	Backward	Hoisting of arithmetic expressions computed on multiple paths through the program	(1) if($x < 0$){
				(2) $y = (z*3)+1$;
				(3) }else{
				(4) $y = (z*3)+2$;
				(5) }
				The expression ($z*3$) is computed at lines (2) and (4). It may make sense to move the computation before line (1).

Abstract Interpretation

Abstract interpretation is a generic term for a family of static analysis techniques that includes both type systems and data flow analysis, among others.[8] In abstract interpretation, a program's variables are assigned values from an abstract domain, and the program is executed on the basis of modified semantics for how each language construct applies in this new domain. For example, whereas an `int` variable may typically take on any concrete integer value in the range `[-2^31, 2^31)`, in abstract interpretation the variable may be given a value of `-, 0, +, or ?` indicating only the sign of the variable (where "?" indicates an unknown or indeterminate value). Operators such as < are given meaning for this new domain, as shown in Table 2.

Moving away from concrete values and operators allows automated analysis tools to evaluate programs' meanings in terms of the higher-level abstract domains. This ensures that the analysis will actually terminate on all input programs.

Abstract interpretation is a powerful tool in program analysis because it can be used to verify many important program correctness properties, including memory, type, and information-flow safety. The primary chal-

Table 2. Abstract interpretation rules for the < operator over the domain `{–, 0, +, ?}`

Input 1	Operator	Input 2	Is	Result
–	<	–	≡	?
–	<	0	≡	True
–	<	+	≡	True
–	<	?	≡	?
0	<	–	≡	False
0	<	0	≡	False
0	<	+	≡	True
0	<	?	≡	?
+	<	–	≡	False
+	<	0	≡	False
+	<	+	≡	?
+	<	?	≡	?
?	<	–	≡	?
?	<	0	≡	?
?	<	+	≡	?
?	<	?	≡	?

Table 3. The lines of code, number of defects reported, and average number of lines per defect for five popular open-source projects as reported by the Klocwork Insight static analysis tool with default settings

Project	Lines of Code	Defects Reported	Lines of Code per Defect
dnsmasq2.51	15,838	20	791.9
util-linux	70,069	2,981	23.5
libxml2	178,406	291	613.1
openssl	189,032	488	387.4
glib	333,884	678	492.5

lenge to applying abstract interpretation is the design of the abstract domain of reasoning. If the domain is too abstract, then precision is lost, resulting in valid programs being rejected. If the domain is too concrete, then analysis may become computationally infeasible.

From Alerts to Alarms

Although automated analysis tools provide a powerful approach to finding potential defects in a large software project, they may easily overwhelm the user with alert messages of questionable importance. For a large code base, a tool may report hundreds or even thousands of alerts, many of which may be spurious or of little impact to the mission of the software. Further, many of the alerts reported may be symptoms of the same underlying error. Table 3 shows the lines of code, number of defects, and lines of code per defect reported by a commercial static analysis tool, Klocwork Insight, that uses a combination of syntax matching and data flow analysis on five popular open-source projects: dnsmasq, util-linux, libxml2, openssl, and glib. These projects vary widely in scope and size but are all fairly mature. Although dnsmasq's 20 reported defects may be easily triaged and addressed, the 2,981 defects reported for util-linux are quite daunting.

Like most static analysis tools, Klocwork Insight assigns each defect a severity based on the type of error reported; for example, buffer overflows are labeled as "Critical" because of the likely security implications, whereas memory leaks are labeled as "Errors" because, although undesirable, they are unlikely to result in security issues. However, the true criticality of a buffer overflow depends on context. For example, a buffer overflow in the configuration parsing code of a web server is probably less severe than a similar error in its URL parsing. Because the configuration file being processed is assumed to be owned by the system administrator running the server, an error in this parsing code would not lead to a privilege execution vulnerability (although it is still a problem and should certainly be corrected). Conversely, the URLs parsed by a web server are provided by an unknown and potentially malicious third party. Thus, a flaw in this code could result in a remote code execution vulnerability that must be addressed at once. Without an understanding of the software's architecture and usage model, static analysis tools are unable to make these kinds of determinations.

The SARA Lab hopes to address these shortcomings of static analysis tools by developing enrichment and visualization tools to help quickly determine the impact of reported defects on an overall software architecture. A first attempt at this is the use of the Dagger visualization

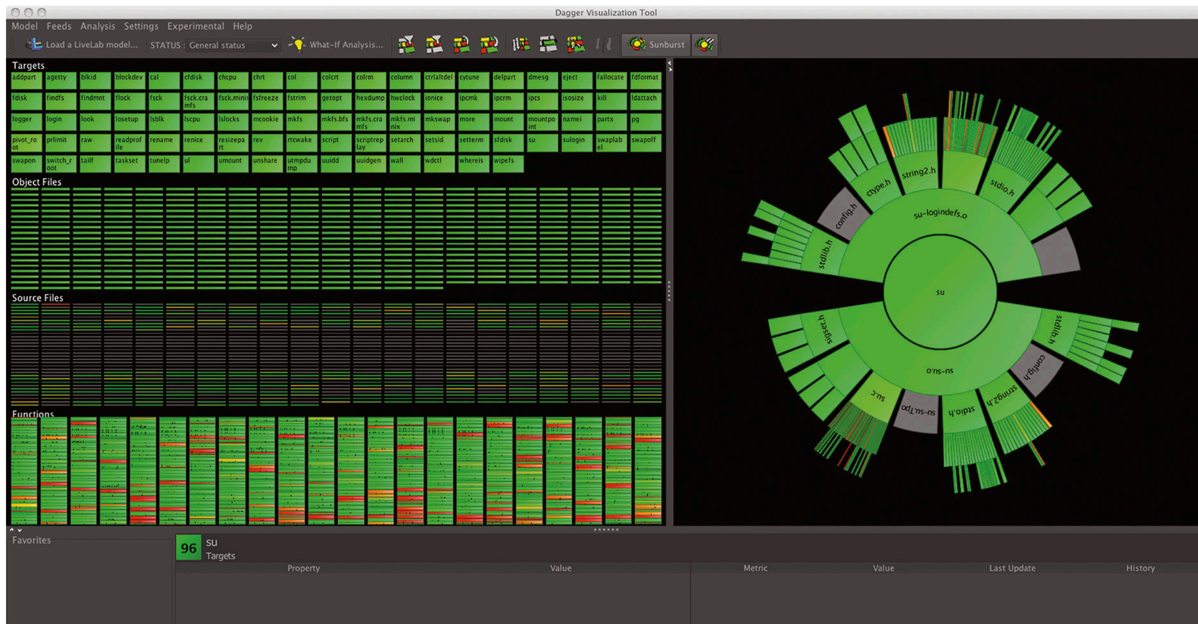

Figure 1. Visualization of the util-linux software architecture and the impact of defects reported by Klocwork Insight.

software, developed for the APL LiveLab to visualize the mission impact of cyber events on a large network, to visualize the dependencies of a piece of software, and to show how defects propagate through the architecture. Figure 1 is a screenshot of the Dagger tool being used to visualize the architecture of the util-linux project and the impact of the defects reported by Klocwork Insight.

The architecture model is automatically generated from an instrumented build of the project and represents the architecture in four tiers: functions, source files, object files, and executables. Any subroutines called by a function are modeled as dependencies of the caller. Source files depend on the function they define, object files depend on the sources that are compiled to create them, and executables depend on the object files linked into them. This model was chosen to provide a coarse, but fully automated, approximation of the conceptual hierarchy of dependencies in the target program. Automatic and user-guided extraction of richer and more semantically meaningful software architecture information is an ongoing area of research in the SARA Lab. The user can refine this architecture by taking advantage of Dagger's built-in functionality for editing the visualized dependency model.

Understanding how best to make use of static analysis results to improve the efficiency of developers in creating correct software is a key part of the SARA Lab's mission to improve the usefulness of static analysis. Analytics and visualizations that incorporate developer understanding can help reflect the security and reliability impact of defects. The context and quick feedback that these tools provide give developers a better chance of managing the sometimes overwhelming sets of alerts for their projects.

THE SARA LAB

In this day and age, when adversaries are searching for any error they might use to their advantage, APL needs to employ every tool at its disposal to prevent delivery of vulnerable software to its sponsors. The SARA Lab was created in summer 2012 to bring existing tools such as those described in the preceding section to APL and to help APL software developers apply them to our software products. As a by-product of this endeavor, we expect that APL will begin to add to the repertoire of technologies and techniques that will ultimately lead to more robust, adversary-proof products.

The SARA Lab Vision

Figure 2 depicts a vision for the SARA Lab as a mature facility at APL. It shows three environments that will coexist and mutually support one another. The research environment at the far left will accommodate the development of novel tools to improve our ability to find defects in complex software and enhance the user experience. Researchers will be inspired by the gaps they experience while working with existing tools; they will draw on an extensive library of real-life, complex APL software products that have been analyzed in the SARA Lab at some previous time. These will provide both a realistic challenge to any new tool and at least partial ground truth because the analysis results for all of the samples in the inventory will available for comparison.

Once a research tool proves its worth on the basis of test data and has achieved some level of development stability, it will be moved to the development environment. The development environment is the heart of the SARA Lab. In this environment, current software projects at APL will be subject to analysis by one or more commercial, open-source, or mature research tools. In the development environment, the software developers work together with SARA Lab researchers to learn how to apply analysis tools and interpret the results. We expect that the SARA researchers will often learn as much as the developers about the tool use and capabilities. This experience will feed the software analysis research that is done in the SARA Lab.

Although many commercial and open-source tools work only on code, FM can verify executable specifications or designs of algorithms cast in a logic language. An APL software product may appear in the SARA Lab multiple times at different points in its development—requirements gathering, design, or implementation—to undergo the appropriate analysis for that stage of development. The lessons learned from the analysis go back into the project to improve the product. The product itself becomes part of the SARA Lab inventory of test samples.

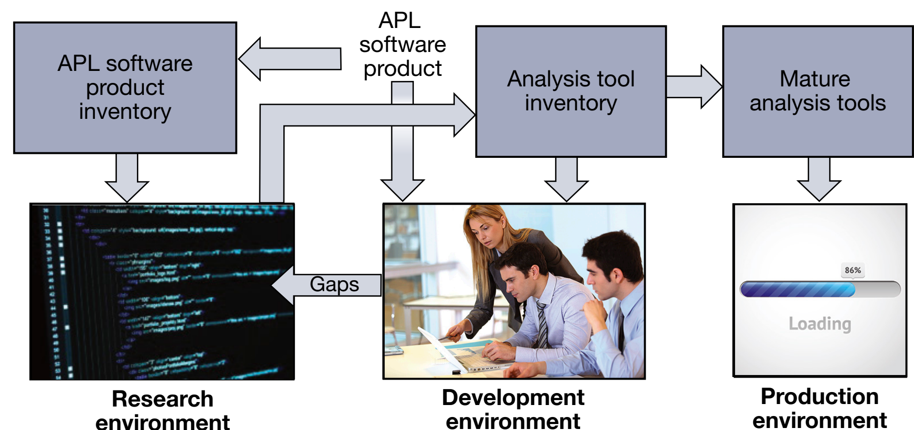

Figure 2. Mature SARA Lab vision.

Finally, when a tool in the development environment proves to be so useful and user-friendly that the support of the SARA Lab researchers is no longer needed, the tool is moved to the production environment. From the production environment, the tool and its documentation can be downloaded for use throughout APL, rather than just in the SARA Lab. While SARA researchers will still support the product, it is envisioned that any tool in the production environment will have a sufficiently large set of practitioners outside of the SARA Lab who can mentor less-experienced users on their team. New tools developed at APL could be open-sourced, to engage the software analysis community in their further improvement.

The First Year of Operation

During its first year of operation, the SARA Lab team focused on building the development environment. Today, it has a significant inventory of tools and an impressive set of early successes.

Tools in the SARA Lab

To kick off operations in the SARA Lab, the SARA Lab researchers explored the available open-source and commercial software analysis tools. A number of these tools were brought into the SARA Lab for experimentation. Figure 3 depicts the result of comparing the output (number of defects found) from four different static analysis tools analyzing the same piece of software. Notably, the overlap in defects found is much smaller than the number of unique defects found by each tool. Two of the tools in this study (the products from Coverity and Parasoft) both use combinations of pattern matching and data flow analysis but had no overlap in the defects found.

To investigate this further, the SARA researchers used a graphical analysis tool (Pointillist) that displays the relationships among various nodes; in this application, the nodes are source lines of code (SLOCs) leading to a defect. Figure 4 is a small part of a graph generated from the output of the same four tools shown in Fig. 3. The parallelogram-shaped nodes represent a SLOC where a defect was detected. The triangular nodes are SLOCs that are not themselves defects but that are on the control path to the defect. The directionality of the edges represents the direction of the control flow. The color of the nodes reflects which tool identified that particular line of code. Although this subgraph shows six defects, the relatively small number of control paths leading to them suggests that the "unique" defects are

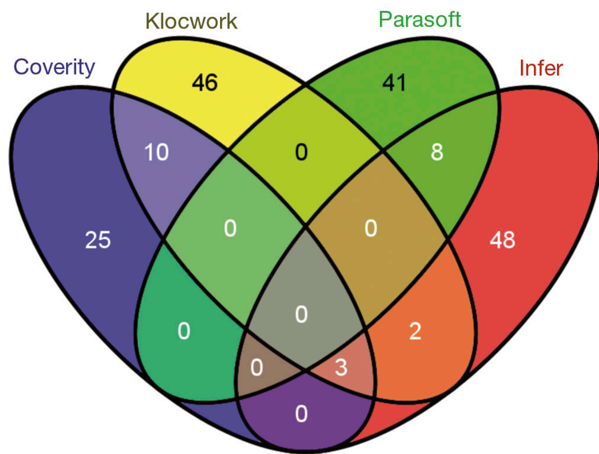

Figure 3. Comparison of bugs found by various static analysis tools.

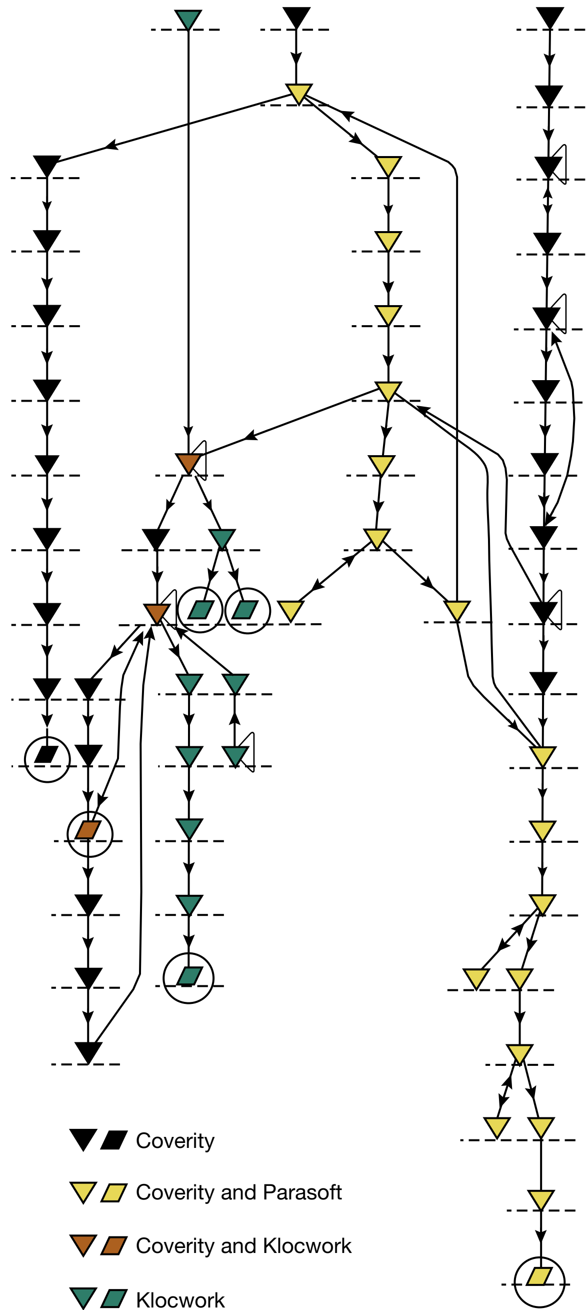

Figure 4. A Pointillist view of defects generated by four analysis tools.

Table 4. The SARA Lab complement of static analysis tools at the time of publication

Tool (source)	Capability
Cppcheck (open source)	Static analysis based on syntactic pattern matching for C and C++ programs
FindBugs (open source)	Static analysis based on pattern matching for Java programs
CQual (open source)	Type system for C programs, enhanced with annotations to provide flow guarantees
Klocwork Insight (commercial)	Static analysis based on both pattern matching and data flow analysis for C, C++, and Java programs
Monoidics Infer (commercial)	Safety analysis based on automated deductive reasoning for C and C++ programs
SPIN (open source)	Model checker

really different manifestations of the same questionable programming construct in the control path.

In addition to the static analysis tools, the SARA Lab has two tools that implement FM analysis approaches. The first is a model checker call SPIN. The second is the Infer tool developed by Monoidics Inc. that applies deductive verification techniques to detect memory violations in C/C++ programs. Unlike most deductive verifiers, the Infer tool is fully automated but only considers, or in FM terminology "reasons about," memory violations; it does not consider other hypotheses that the analyst may wish to explore.

The complete set of static analysis tools available in the SARA Lab at the time of this publication is given in Table 4. Within budgetary constraints (some commercial tools are quite expensive), the SARA researchers attempted to get the best coverage of techniques within the tool set.

Completed Analyses

To date, both the Asymmetric Operations Department and the Space Department have submitted products for analysis. Over the time of the SARA Lab's operation, the lab's researchers have analyzed thirteen software products and the Trusted Computing Group's Trusted Platform Module 2.0 executable specification. This analysis has uncovered a grand total of 12,008 defects and resulted in updates to five of the analyzed products. All of the developers who have had their products analyzed in the SARA Lab had a very positive reaction to the process. The individual software products that have been analyzed are given in Table 5, along with the results of the analysis.

Table 5. Results for software products analyzed in the SARA Lab

Software Product	Language	No. of Defects	No. of Defect Fixes Due to Analysis
Linux Kernel Integrity Monitor	C	996	896
Userspace Measurement	C	910	723
Libspd-userspace	C	894	888
TPM 2.0 Executable Specification	C	67	30
Network Control Software	Java	536	0
Ruby Mirror	C++	34	0
Alien Autopsy	Java	122	0
LAMP	Java	6,724	2,100
gmaplib	C++	597	0
TDSS	C	42	0
Mission-based analysis	Java	62	0
Pointillist	Java	1,024	0

Of note are the first four products in Table 5. These four products are intended to perform security functions; that is, that is, they are being trusted to ensure the correct operation of a platform or network that may come under attack. To achieve trustworthy computing, the security applications, at a minimum, must be demonstrated to be error free (where an error may introduce a vulnerability).

Using the SARA Lab

Because one of the goals of the SARA Lab is to improve the software products delivered by APL to its sponsors, analysis in the SARA Lab is open to all APL staff. A self-serve web portal where developers can submit software products for analysis is under construction. Today, the best approach to getting a product into the SARA Lab is to make contact with the researchers by sending an e-mail to +Saralab. SARA Lab researchers will then schedule a meeting to discuss the project needs and suggest an approach for meeting those needs.

Interested software developers at APL are also invited to join the SARA Lab Cooler group "Software Analysis Research and Applications." In this group, participants share information about topics pertaining to static analysis, including tool surveys, screenshots of tools to demonstrate their use, descriptions of research advances in static analysis, and postmortems of software vulnerabilities that could have been detected by using static analysis.

CONCLUSION

Securing computing resources against malicious attack is not simply a research topic at APL. Instead, it is a goal that we wish to achieve throughout the national

security community. We can begin that process "at home," by bringing APL software products into an environment where the best, state-of-the-art tools for ferreting out potential errors—and the vulnerabilities they may create—are used. The SARA Lab is providing that environment. Although the SARA Lab has not been in existence for very long, it has already made a difference in APL software products. When the full vision for the SARA Lab is realized, it should be a unique resource for APL and a model for others in the national security research and development community.

REFERENCES

[1]Johnson, S. C., "Lint, a C Program Checker," in *Bell Laboratories Computer Science Technical Report 65*, pp. 78–1273 (1977).

[2]Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, MA (2002).

[3]Novak, G. S., "Conversion of Units of Measurement," *IEEE Trans. Softw. Eng.* **21**(8), 651–661 (1995).

[4]Shankar, U., Talwar, K., Foster, J. S., and Wagner, D., "Detecting Format String Vulnerabilities with Type Qualifiers," in *Proc. 10th Conf. on USENIX Security Symp.*, Berkeley, CA, Vol. 10, p. 16 (2001).

[5]Pratikakis, P., Foster, J. S., and Hicks, M., "Locksmith: Context-Sensitive Correlation Analysis for Race Detection," in *Proc. 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, pp. 320–331 (2006).

[6]The Coq Development Team, *The Coq Reference Manual version 8.4*, http://coq.inria.fr/distrib/current/refman/.

[7]Kildall, G., "A Unified Approach to Global Program Optimization," in *Proc. 1st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, New York, pp. 194–206 (1973).

[8]Cousot, P., and Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Conf. Record of the Fourth ACM Symp. on Principles of Programming Languages*, pp. 238–252 (1977).

The Authors

J. Aaron Pendergrass is the Principal Investigator (PI) for the SARA Lab and a computer scientist in the APL's Asymmetric Operations Department (AOD). He began development of the SARA Lab in order to centralize APL efforts to improve the software engineering process and the assurance of critical software correctness. As the Chief Scientist of AOD, **Susan C. Lee** develops system concepts, technology roadmaps, and sponsor engagement strategies across the cyber operations, special operations, national security, and homeland protection domains. Her domain knowledge includes intelligence, information operations, radar propagation, spacecraft development and operation, and medical devices. Her vision for the future of software assurance and understanding of the real-world challenges have shaped the short-term and long-term strategies for development of the SARA Lab. **C. Durward McDonell** is a computer scientist in AOD with a background in mathematics. As a direct technical contributor to the SARA Lab, Dr. McDonell's primary interest is in formal methods and the mathematics of software. The SARA Lab is intended as a resource to advance the state of the art and improve software assurance both within APL and beyond. For further information on the work reported here, contact J. Aaron Pendergrass. His e-mail address is aaron.pendergrass@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.