ART-006: Risk Based Approach to Cyber Vulnerability Assessment using Static Analysis

Sponsor: OUSD(R&E) | CCDC AC

Dr. Peter Beling, Dr. Stephen Adams, Mr. Tim Sherburne, Mr. Davis Loose

Research Team: Dr. Stephen Adams, Dr. Peter Beling (PI), Dr. Barry Horowitz, Mr. Davis Loose, Mr. Tim Sherburne

University of Virginia

12th Annual SERC Sponsor Research Review
HOSTED VIRTUALLY ON: November 18, 2020

www.sercuarc.org
Background and Objectives
A significant number of static analysis tools exist for discovering cyber security vulnerabilities supporting a variety of:

- Operating Systems, including Linux and Windows
- Programming and scripting languages, including C/C++, Java, Perl and Python

Tools discover SW structures that can support the objectives of cyber attackers, and provide input to programmers and designers for consideration regarding modification of the identified vulnerable SW.

Alerts are principal tool outputs – identification of specific lines of code that are problematic and should be candidates for modification.
DoD user experience has identified significant productivity issues with using Static Analysis tools for cyber security purpose:

- The large number of alerts
  - Most alerts are **false positives**
  - Alerts can number in the 10’s of thousands
- Time and analyst skill level needed to correct true positives
  - Fix identified code vulnerability
  - Expand to related or similar code?
  - Other defensive measures or resilience?
Source Code Analysis Lab (SCALe)
SEI / CCDC AC (DEVCOM)

https://github.com/cmu-sei/SCALe
1. Extend the static analysis capabilities of the CCDC AC Software Assurance Toolset by integrating filters based on functional risk.
   - Leverage Cyber Security Requirements Methodology (CSRM) previously developed in collaboration with CCDC AC
   - WRT-1013, ART-004/WRT-1033, WRT-1022

2. Develop an optimization framework and associated machine learning methods to further improve the efficiency of the static analysis process
Source Code Analysis Lab (SCALe)

Alert Prioritization Techniques

https://github.com/cmu-sei/SCALe
Mission Priority using Dynamic Tracing
Problem:

For the consequences to be avoided, which are dependent on which lines of code identified via static analysis?
• Dynamic system testing currently includes tests that relate to other than cyber-attack stimulants for system fault tolerance and resilience features:
  — Safety
  — Operator errors
  — Out of Spec situations (e.g., overloads, potential anomalous circumstances)
  — System countermeasures (electronic warfare, tampering)
  — Technology component failures (hard and soft failures)
  — Etc.

• These focused dynamic system tests provide a basis to use already available compiler-based SW tracing results as a means for identifying the specific SW modules and files used in the process of evaluating system design related to specific faults
Hypothesis regarding application of static analysis results to prioritization of cyber attack risks:

- System risks which are currently tested for will include significant consequence overlaps with those derived from cyber attacks, thereby providing a basis for using SW tracing based upon already existing system tests as a mechanism for identifying which of the static analysis results relate to which of the identified cyber attack consequences.

- As cyber attack resilience emerges as an additional area of system design, dynamic system test results for cyber resilience can be utilized to enable more effective use of new static analysis results that emerge over the life cycle due to:
  - system design changes,
  - changing cyber attack techniques,
  - new findings that result from modifications in static analysis tool designs.
Mission Aware Metamodel (UVA)

MBSE

STPA

Mission Aware

MA MBSE Meta-Model

Mission Aware
Cyber-Security Requirements Methodology (CSRM)

- What to protect and why? Which combination of design patterns to employ in which mission subsystems?
- Standard Blue Team (Mission), Gold Team (SE), Red Team (Threat) methodology for evaluating resilience with models
STPA is an iterative, methodical **hazard analysis technique** to identify causes of hazardous conditions intended to improve or promote **system safety**.
- In cyber-physical systems, **security** can be treated as analogous to safety.

**STPA Outputs and Traceability**

*Figure 2.21* shows the traceability that is maintained between various STPA outputs.

- **A Loss** involves **something of value** to stakeholders. Losses may include a loss of human life or human injury, property damage, environmental pollution, loss of mission, loss of reputation, loss or leak of sensitive information, or any other loss that is unacceptable to the stakeholders.
- **A Hazard** is a **system state** or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to a loss.
- An **Unsafe Control Action** (UCA) is a control action that, in a particular context and worst-case environment, will lead to a hazard.
- **A Loss Scenario** describes the **causal factors** that can lead to the unsafe control and to hazards.
1. The Cyber-Security Requirements Methodology is applied to the System being analyzed in order to determine a priority-based set of consequences to be avoided with associated system resilient modes.

2. A set of system anomaly cases are defined that would ultimately serve to test the mission priority resilient modes as identified by step 1.

3. The system software is evaluated by one or more static analysis tools in order to identify software vulnerabilities (referred to as alerts).

4. The system software is compiled with dynamic tracing enabled.

5. The system anomaly cases identified in step 2 are executed with tracing enabled. For each test case, a trace file is collected that identifies the functions and lines of code executed during the test.

6. The static analysis results (step 3) are integrated with the anomaly case traces (step 5). This integration provides a mechanism of prioritizing static analysis alerts based on the priority of any associated test traces for the identified code location (file name / line number).

7. An Analyst evaluates the integrated set of Static Analysis Alerts using the Resilient Mode System Test Priority as a mechanism to sort and filter the alert list.
ArduPilot is an open source autopilot system supporting multiple autonomous vehicle types.

**SITL** (software in the loop) simulator allows ArduPilot execution without vehicle hardware.

ArduPilot’s **Auto Test** suite allows for the creation of repeatable tests of autopilot behavior based on SITL simulator.
Led by the Morgridge Institute for Research in Madison WI, the Software Assurance Marketplace (SWAMP) is a no-cost, cloud service that provides Static Code Analysis to developers and researchers.

<table>
<thead>
<tr>
<th>Available SA Tools (C/C++)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td>Open Source</td>
</tr>
<tr>
<td>Cppcheck</td>
<td>Open Source</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>Commercial</td>
</tr>
<tr>
<td>Coverity</td>
<td>Commercial</td>
</tr>
<tr>
<td>Code DX (Consolidated Results Viewer)</td>
<td>Commercial</td>
</tr>
</tbody>
</table>

- Variation in SA tool results are similar to findings of NIST Static Analysis Tool Exposition (SATE).
- Alert Density of 2.0 equates to 2,000 alerts for a 1 million LOC project.

https://www.mir-swamp.org/
https://scanCOVERITY.com/

<table>
<thead>
<tr>
<th>Code DX Results Summary</th>
<th>Alert Count</th>
<th>Alert Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td>439</td>
<td>2.02</td>
</tr>
<tr>
<td>Tool B</td>
<td>778</td>
<td>3.59</td>
</tr>
<tr>
<td>Tool C</td>
<td>66</td>
<td>.30</td>
</tr>
<tr>
<td>Tool D</td>
<td>43</td>
<td>.20</td>
</tr>
<tr>
<td><strong>(ArduPilot) Total Alerts</strong></td>
<td><strong>1,326</strong></td>
<td></td>
</tr>
</tbody>
</table>

1. Licensing terms prevent publication of tool specific results (list is reordered)
2. Little to no overlap between tools
3. Alerts / 1,000 Lines of Code (LOC)
4. Over full code base
    (all vehicle modes, test drivers, link protocol libs, etc.)
Code is instrumented at *Compile-time* to output function / line execution counts per code file at *Run-time*.

Each Test Case captures a Coverage Report

![Coverage Report Diagram]

**ArduPilot Helicopter SITL build includes:**
- **7,546 Functions**
- **69,743 LOC**  

**Merged Coverage Results for “Common” Code Executed Across Test Cases.**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Test Type</th>
<th>Functions Executed</th>
<th>LOC Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loiter to Altitude</td>
<td>Safety</td>
<td>533</td>
<td>4291</td>
</tr>
<tr>
<td>Battery Failsafe</td>
<td>Component Failure</td>
<td>329</td>
<td>2927</td>
</tr>
<tr>
<td>Camera Control</td>
<td>Component Failure</td>
<td>391</td>
<td>3377</td>
</tr>
<tr>
<td>GPS Glitch</td>
<td>Out of Spec</td>
<td>373</td>
<td>3287</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Common Across:</th>
<th>Functions Executed</th>
<th>LOC Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(or more) Test Cases</td>
<td>376</td>
<td>3319</td>
</tr>
<tr>
<td>(or more) Test Cases</td>
<td>343</td>
<td>2914</td>
</tr>
<tr>
<td>4 Test Cases</td>
<td>245</td>
<td>2260</td>
</tr>
</tbody>
</table>

---

1 Does not include `#define`, `#if <def>` compiler directive, test code
Static Analysis Alerts are Attenuated by Correlation to High Priority Mission Test Case Code Files (4 Static Analysis Tools with 4 Test Cases)

<table>
<thead>
<tr>
<th>SA Tool</th>
<th>Total Alerts</th>
<th>ArduPilot SA Alerts Correlated to Code Coverage Test Files</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Alerts found in Files Executed in at least 1 Test Case</td>
<td>Attenuation</td>
</tr>
<tr>
<td>Tool A</td>
<td>439</td>
<td>91</td>
<td>79%</td>
</tr>
<tr>
<td>Tool B</td>
<td>778</td>
<td>204</td>
<td>74%</td>
</tr>
<tr>
<td>Tool C</td>
<td>66</td>
<td>3</td>
<td>95%</td>
</tr>
<tr>
<td>Tool D</td>
<td>43</td>
<td>12</td>
<td>72%</td>
</tr>
<tr>
<td>Total</td>
<td>1,326</td>
<td>310</td>
<td>77%</td>
</tr>
</tbody>
</table>

Initial Findings:

- For ArduPilot, filtering alerts based on Mission Priority test cases, provides attenuation of ~ 75%
- A small number of test cases can provide reasonable alert filtering results
1. SWAMP-based SA Importer: A proof-of-concept importer for SWAMP static analysis tools based on “SWAMP Common Assessment Result Format” (SCARF).
   - tools.org
   - swamp-clang2org.py
   - c_swamp-clang.properties
   - swamp-codesonar2org.py
   - c_swamp-codesonar.properties

2. GCOV to MPCS Script: A script to summarize multiple GCOV trace files into a Mission Priority Code Summary JSON document
   - gcov2mpcs.py

3. MPCS Importer: A script to import the mission-priority.json document. The script updates the alert database (see Grid UI) and the code navigation html.
   - mission-priority2sql.py

4. Extend DB for Mission Priority: A new DB column to store the highest (6 is highest) associated mission priority for an alert. The mission priority has the following values:
   - 0: The alert is not associated with a mission priority case.
   - 1: The alert is associated with a single low mission priority case.
   - 2: The alert is associated with multiple low mission priority cases.
   - 3: The alert is associated with a single medium mission priority case.
   - 4: The alert is associated with multiple medium (or lower) mission priority cases.
   - 5: The alert is associated with a single high mission priority case
   - 6: The alert is associated with multiple high (or lower) mission priority cases.
SCALe Demo
Machine Learning for Alert Prioritization
Addressing static analysis alerts is an expensive and time-consuming process. This is exacerbated by other factors such as false-positive alerts. As such, it is useful to develop techniques to efficiently address the set of alerts and improve productivity.

The CAP Methodology is a high-level framework for prioritizing static analysis alerts. There are many ways to accomplish this (such as RSVAM, which filters low priority alerts), but the objective is always to determine how to distribute effort given limited resources.

**Collect**

*Gather data and other models, including:*

- Source Code Files
- Static Analysis Alerts
- Requirements and Other Documentation
- Language Models
- Code Coverage Reports
- RSVAM

**Assess**

*Apply human logic and statistical models to data, including:*

- Classifiers and Associated Models
- Code Criticality Metrics
- Times to Resolve and Other Cost Metrics
- Binary Classification of Top Priority Alerts

**Prioritize**

*Optimize, sort, and filter alerts through processes such as:*

- Confidence in True/False Positives
- Optimization
- Organization by Value Added Per Unit Time
- Cost or Time Limitations
- Filter Low Priority Alerts
The following slides present an example of the CAP Methodology using ArduPilot. The ArduPilot case study uses topic modeling, simulated resolution times and classification, and optimization to filter and prioritize alerts.
Topic modeling is a machine learning technique for classifying texts based on latent themes

- Gathered ArduPilot static analysis results from SCALe and develop a topic model
- Develop topic model:
  1. Preprocess ArduPilot code for training data
  2. Develop topics using the latent Dirichlet allocation model
  3. Observe topics and rank in order of importance
  4. Ascribe weights to each topic – for this example, [1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]

<table>
<thead>
<tr>
<th>Topic #</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic 1</td>
<td>Navigation Control</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Movement Calculations</td>
</tr>
<tr>
<td>Topic 3</td>
<td>Communication</td>
</tr>
<tr>
<td>Topic 4</td>
<td>Checks and Failsafes</td>
</tr>
<tr>
<td>Topic 5</td>
<td>Path Planning</td>
</tr>
<tr>
<td>Topic 6</td>
<td>Mission Management</td>
</tr>
<tr>
<td>Topic 7</td>
<td>Sensors</td>
</tr>
<tr>
<td>Topic 8</td>
<td>Parameter Storage</td>
</tr>
<tr>
<td>Topic 9</td>
<td>SITL Modules</td>
</tr>
<tr>
<td>Topic 10</td>
<td>Various</td>
</tr>
</tbody>
</table>
• Simulated a classifier to assign true-positive likelihoods for each alert
  • Based on existing research, each alert is assigned a true-positive likelihood based on known distributions

• Simulated times to resolve
  • Each static analysis alert is assigned a time to resolve, simulated from a known distribution
  • Each alert received two times: one for a true-positive result, and another for a false-positive result
• Used knapsack optimization to maximize value added given limited time resources
  • The value of each alert is provided by the static analysis tools as the priority score, scored 1-27
  • This score is weighted by the topic mixture of the code module containing each alert (less important topics may reduce the value of an alert)

• Organized the alerts by value per unit cost, in this case value per hour

• Right: CAP is compared to the base method in which a user addresses the highest priority alerts first, as scored by the tools in SCALe
• Compared to other statistical techniques used in CAP, the true-positive classifier has an outsized impact on performance
  • Right: As the true-positive threshold rises, confidence in classifier performance decreases
  • With lower confidence in the classifier, the net value added by CAP compared to the base method is decreased

• It is critical to develop robust alert classification models to maximize the impact of CAP
Limitations of Static Analysis

• No single tool dominates the market
  — Developers generally use multiple tools

• Generate many alerts
  — When using multiple tools, sets of alerts can be disjoint because different tools focus on different types of software defects

• False alarms

• Low-priority alerts

Possible solution

Use machine learning techniques to improve efficiency
• Predict True Positive or False Positive label in order to triage alerts

• Binary Classification

• CNNs

• Classifier comparison
Labeling Static Analysis Alerts

- Supervised ML requires a large and accurately labeled dataset
- Time consuming
- Requires training

- A change in the static analysis tool set could require the labeling of a new data set
  - Addition or subtraction of a tool
  - Upgrade of a tool
• Use active learning to reduce the number of labeled examples needed to train a classifier

• Evaluate active learning algorithms
Active Learning for Static Analysis

1. Run static analysis tools on source code
2. Extract features from source code and static analysis results
3. Select initial training set
4. Label training set using auditor
5. Learn binary classifier
6. Use active learning to select new examples and add to training set
• Juliet test suite
  — Publicly available by NIST
  — C/C++ and Java
  — Examples of Common Weakness Enumerations (CWEs)
  — Manifest of CWEs

• Run 2 static analysis tools

• Use manifest to label alerts generated by the tools

• Produces data set of ~60K observations with 17% true positive rate

• Publicly available on GitHub

### Data Set

<table>
<thead>
<tr>
<th>Feature Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severity</td>
<td>Severity by Tool A</td>
</tr>
<tr>
<td>CWE</td>
<td>Mapping to CWE by either tool</td>
</tr>
<tr>
<td>Tool A Alert</td>
<td>Binary – alert by tool A</td>
</tr>
<tr>
<td>Tool B Alert</td>
<td>Binary – alert by tool B</td>
</tr>
<tr>
<td>Tool A Rule</td>
<td>Rule output for tool A</td>
</tr>
<tr>
<td>Tool B Rule</td>
<td>Rule output for tool B</td>
</tr>
<tr>
<td>Line</td>
<td>Line of code for alert</td>
</tr>
</tbody>
</table>
Results

![Graph 1: Performance vs. Number of Queries](image1)

- blue line: uncertainty
- orange line: by committee
- green line: random

![Graph 2: Performance vs. Number of Queries](image2)

- blue line: uncertainty
- orange line: by committee
- green line: random
• Developed Risk-based Software Cyber Vulnerability Assessment Methodology (RSVAM)
  — RSVAM combines the Cyber-Security Requirements Methodology (CSRM) approach to function risk analysis with dynamic code tracing.
  — Software for RSVAM was developed as an extension to the SCALe open source GitHub release (2.5.1.0 – Nov2018)

• Developed the CAP methodology framework for prioritizing static analysis alerts

• Demonstrated that active learning can be used to reduce the amount of data needed to train a classifier for detecting true positive SA alerts

• Limitations
  — ARDU Pilot may not represent complexity or challenges of Army SW systems
  — Active learning methods trained on an artificial code base
Backup
SCALe User Interface

MISSION CASES - Associated Alerts (22 of 115 - 80% Attenuation)

High
- Latching to Altitude
- Remote Device Failure
- Medium
- Battery Failures
- Low
- OFP Clinch
- Camera Control

MAINS
- main: archpilot/Tools/AP_Bootloader/AP_Bootloader.cpp int main(void)