AFRL-RI-RS-TR-2019-232

RULER: A SYSTEM TO DETECT RESOURCE USAGE VULNERABILITIES WITHIN SOFTWARE

IOWA STATE UNIVERSITY

DECEMBER 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-232 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WALTER S. KARAS
Work Unit Manager

/ S /
JAMES S. PERRETTA
Deputy Chief, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2019	FINAL TECHNICAL REPORT	APR 2015 – JUL 2019

4. TITLE AND SUBTITLE

RULER: A SYSTEM TO DETECT RESOURCE USAGE VULNERABILITIES WITHIN SOFTWARE

5a. CONTRACT NUMBER
FA8750-15-2-0080

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Suraj Kothari

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
IW

5f. WORK UNIT NUMBER
WA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Iowa State University
1350 Beardshear Hall
Ames, IA 50011-2025

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

DARPA/I20
675 North Randolph Street
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-232

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The STAC program enabled us to make significant advances. We consider the foremost advance to be intelligence amplifying (IA) technology for detecting AC and SC vulnerabilities. The report describes the challenges encountered and how we have addressed them. Our performance steadily improved with the advances we made with our approach. We achieved top performance on DARPA challenge engagements. We found the take-home and on-site engagements very useful to drive our research. The engagements enabled us to identify the weaknesses in our approach and provided concrete directions to refine our approach.

15. SUBJECT TERMS

Algorithmic Complexity (AC) Vulnerability, Side Channel (SC) Vulnerability, Intelligence Amplifying Technology for Detecting AC and SC vulnerabilities.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WALTER S. KARAS**
U	U	U	UU	40	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1. Executive Summary

Space-Time Resource Usage (STRU) vulnerabilities are rooted in the space and time complexities of externally controlled execution paths containing loops. The STAC program considers two kinds of STRU vulnerabilities: algorithmic complexity (AC) and side channel (SC) vulnerabilities. AC vulnerabilities involve execution paths with high space or time complexity that are triggered by relatively small inputs. SC vulnerabilities involve a set of execution paths associated with a secret that can be inferred from variations in space or time complexities along those paths.

At the extreme, a completely automated detection of STRU vulnerabilities amounts to a solution of the halting problem [1], an unsolvable problem whose roots go back to the Continuum Hypothesis in 1878 by Cantor [2] that later resulted in the proofs of undecidability and incompleteness of axiomatic mathematics by Gödel [3] and Cohen [4]. The good news is that our adversaries cannot solve the halting problem either. Thus, detecting STRU vulnerabilities is a challenging, but not impossible program analysis endeavor, especially when the pragmatic goal is to surpass the capabilities of the adversaries. Our novel solution RULER (**R**esource **U**sage vu**L**nerability identifiER) pioneers a revolutionary machine-human integration that helps it scale to large software while maintaining accuracy in detecting sophisticated STRU vulnerabilities.

Section 6.1 gives a summary of our results in the STAC engagements. Our success on STAC is directly attributable to the identification of, and novel solutions for, the following research questions:

1. How to mathematically model space-time complexities of novel, sophisticated, and domain-specific STRU vulnerabilities?
2. How to statically compute a program's relevant behaviors to deal with the path explosion problem?
3. How to effectively leverage the strengths of both static and dynamic program analysis to verify and validate software anomalies?
4. How can a machine-human analysis system deal with intractable problems in program analysis to detect novel, sophisticated, and domain-specific STRU vulnerabilities?

To address the above research questions, our research and technical approach to detect sophisticated resource usage vulnerabilities was focused on addressing the inherent program analysis challenges in the following categories:

1. Loop Modeling for AC/SC vulnerabilities
2. Execution Path Sensitive Analysis
3. Composable Accuracy Boundary aware analysis
4. Interaction and Programmable What-if Reasoning

5. High level abstractions to model vulnerability mechanisms

Section 2 gives an overview of the project. Section 3 lists our technical approach. Section 4 enumerates the technological advances we have made in RULER to solve the STAC problem. Section 5 gives details on how we apply the technological advances to solve STAC engagement problems. Section 6 gives a summary of our Engagement results. This is followed by our Concluding remarks in Section 7 and a list of our publications and presentations in Section 8.

2. Introduction

This work was performed by Iowa State University (ISU) and was issued by the Air Force Research Laboratory (AFRL) under Cooperative Agreement No. FA8750-15-2-0080, Space/Time Analysis for Cybersecurity (STAC). Our team is composed of Iowa State University (ISU) as the prime and EnSoft Corp. as a subcontractor, each of which brought a unique capability to this project.

Suraj Kothari, the principal investigator, and his team at Iowa State University have a track record of innovative advancements in applications of program analysis. Wei Le, a co-principal investigator has extensive experience in hybrid analysis of loops. Srikanta Tirthapura is also a co-principal investigator and has experience in scalable graph algorithms and databases. Finally, Jeremias Sauceda, a co-principal investigator and his team at EnSoft have extensive experience building highly scalable static analysis tools.

As described by DARPA STAC BAA:

> *The STAC program seeks new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, specifically vulnerabilities to algorithmic complexity and side channel attacks. STAC seeks to enable analysts to identify algorithmic resource usage vulnerabilities in software at levels of scale and speed great enough to support a methodical search for them in the software upon which the U.S. government, military, and economy depend. STAC seeks new techniques and tools; efforts to merely apply existing techniques and tools to finding space/time vulnerabilities are out of scope.*
>
> *The STAC program seeks advances along two main performance axes: scale and speed. Scale refers to the need for analyses that are capable of considering larger pieces of software, from those that implement network services typically in the range of hundreds of thousands of lines of source code to even larger systems comprising millions or tens of millions of lines of code. Speed refers to the need to increase the rate at which human analysts can analyze software with the help of automated tools, from thousands of lines of code per hour to tens of thousands, hundreds of thousands, or millions of lines of code per hour. The human component of semi-automated analysis time tends to dominate the*

automated component, as typical automated program analysis tools require the human analyst to invest considerable initial effort in preparing the software to examine for consumption by the automated tool and later in distinguishing between reports that represent actual vulnerabilities from those that are merely false alarms. Reducing the burden of manual annotation appears likely to be a key part of increasing speed, as is reducing false alarms (that is, increasing precision) while maintaining an acceptably low missed detection rate (that is, decreasing the cases in which the analysis is unsound).

3. Technical Approach

The Space/Time Analysis for Cybersecurity (STAC) program was designed to identify vulnerabilities related to the space and time resource usage behavior of algorithms in Java bytecode applications. The goals as set in the program BAA are listed below:

1. Develop automated program analysis techniques and tools that a human analyst can use to identify algorithmic resource vulnerabilities in software, with equal attention to vulnerabilities to both algorithmic complexity and side channel attacks in both space, and time-based variations. At levels of scale and speed great enough to support a methodical search for them in the software upon which the U.S. government, military, and economy depend.

2. Conduct novel research that leverages advances in program analysis to develop innovative capabilities, far beyond existing practices, to detect highly sophisticated algorithmic resource attacks that the DoD needs to be prepared for.

3. It is reasonable to expect the human analysts who would ultimately use the kinds of tools developed in the STAC program to have a level of general software development expertise, equal to the developers of the target software being examined. As well as, a level of familiarity with the target software's management and operation equal to that of a skilled user or administrator.

Our high-level technical approach to detect sophisticated resource usage vulnerabilities is driven by the following analysis guidelines:

1. **Full Automation** to discover suspicious program segments.

2. **Human-on-loop Static Analysis** to gather corroborating evidence.

3. **Human-on-loop Dynamic Analysis** to understand dynamic behaviors and generate exploits.

In the next section, we will elaborate on the technological advances for each of the above analysis guidelines.

4. Methods, Assumptions, and Procedures

Our novel solution RULER (**R**esource **U**sage vu**L**nerability identifi**ER**) pioneers a revolutionary machine-human integration that helps it scale to large software while maintaining accuracy in detecting sophisticated STRU vulnerabilities. RULER is built on top of the Atlas platform as a set of Eclipse plug-ins. It incorporates the analysis guidelines we mentioned earlier and contains a set of tools and means to address the inherent program analysis challenges in the following categories:

1. Loop Modeling for AC/SC vulnerabilities
2. Execution Path Sensitive Analysis
3. Composable Accuracy Boundary aware analysis
4. Interactive and Programmable What-if Reasoning
5. High-level abstractions to model vulnerability mechanisms.

4.1 Architecture diagram

RULER system architecture is shown in Figure 1.

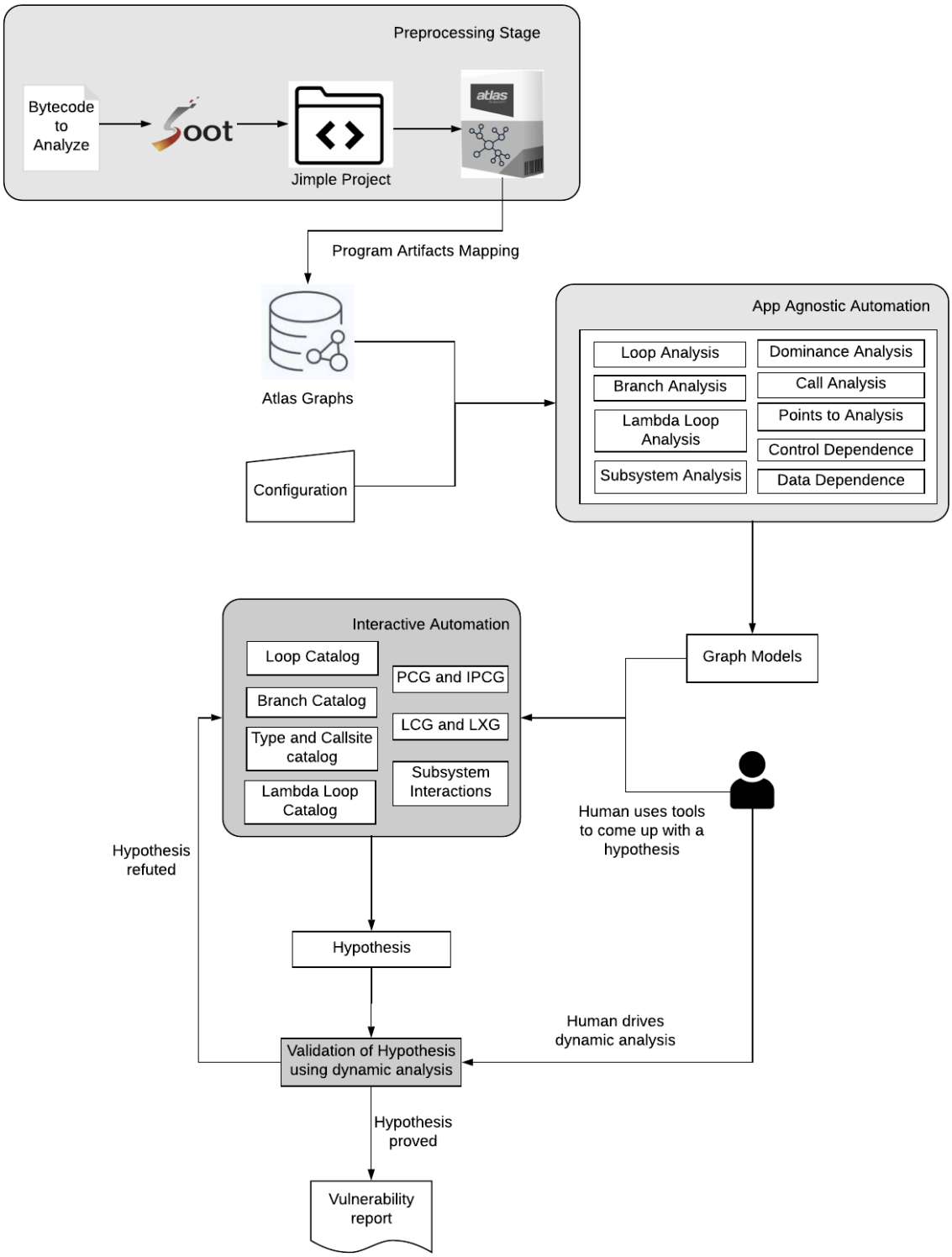

Figure 1. RULER system architecture

4.2 Discover Suspicious Program Segments

4.2.1 Loop Catalog

RULER can detect loops in given Jimple code using the Decompiled Loop Identification Algorithm by Wei et al [5] and the Loop Catalog tabulates the characteristics of each loop. RULER computes loop summaries consisting of 65 attributes. These include

- Source correspondence
- Loop monotonicity
- Loop iteration bounds
- Reachability
- Containment relationships with other loops, branches and lambda loops
- Subsystem interactions
- Path summaries w.r.t call sites (Java Development Kit (JDK) and non-JDK)

Figure 2 shows a screenshot of the Loop Catalog View. Each column corresponds to a characteristic. Each row corresponds to a loop header in the application.

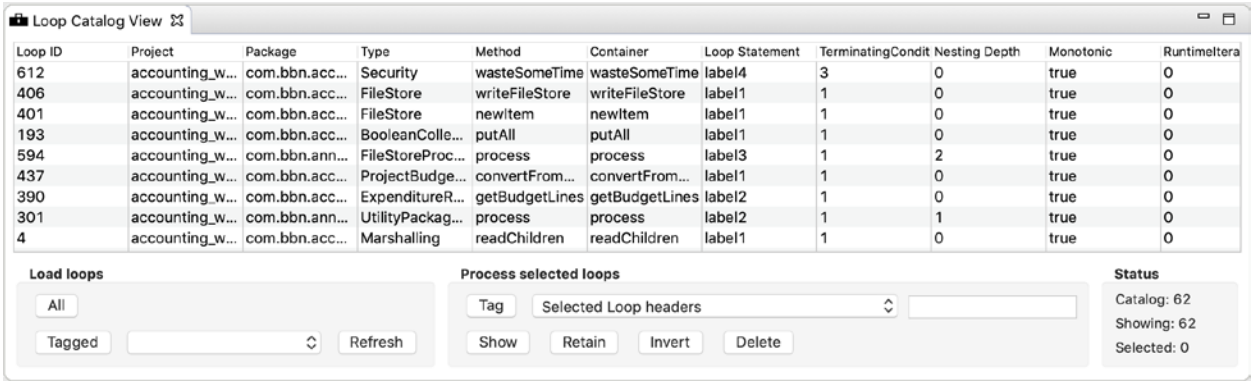

Figure 2. Loop Catalog View

To start using the Loop Catalog, click “All” in the “Load loops” section of the view.

Analysts can click on a column header to sort the loops (rows) with respect to that loop characteristic. Analysts can select a subset of loops, which also selects the corresponding loop header in the Atlas graph, and can be used to drive Smart Views. Selection of a cell under a column representing interaction with a particular subsystem will change the focus of the “Subsystem Interaction” and “Subsystem Interaction PCG” Smart Views to that subsystem.

Loop selections can also be used to control the contents of the view: selected loops may be retained, (soft) deleted, or the selection may be inverted. Analysts can tag selected loops, or load loops with a certain tag. Clicking “All” will reload all loops. This is important to facilitate composition of the loops in the catalog with other analyses and Smart Views.

4.2.2 Branch Catalog

RULER is equipped with the capability to characterize branches in a given app. These characteristics include information about the branch’s location (project, method, etc.), the loops it is contained in or loops it governs, whether it is a termination condition, and call sites to various subsystems that it governs. All the computed characteristics of the branch are saved in the Atlas Graph as eXtensible Common Software Graph (XCSG) tags and attributes.

The Branch Catalog view provides a graphical interface for analysts to browse and filter branches in an app. Similar to the Loop Catalog view, the Branch Catalog view allow analysts to load all branches, or those with a specified tag name. Each row corresponds to a branch, and analysts can click on a column header to sort the rows with respect to that characteristic. Each column corresponds to a characteristic; in total 44 characteristics are provided.

Selection of branches in the table also selects the branch in the Atlas graph, which the Smart Views respond. Selection of a cell under a column representing interaction with a particular subsystem will change the focus of the “Subsystem Interaction” and “Subsystem Interaction PCG” Smart Views to that subsystem. Analysts can manage the contents of the view by selecting a subset of branches, and may retain or delete the selection from the view, or invert the selection.

Figure 3 shows a screenshot of the Branch Catalog View.

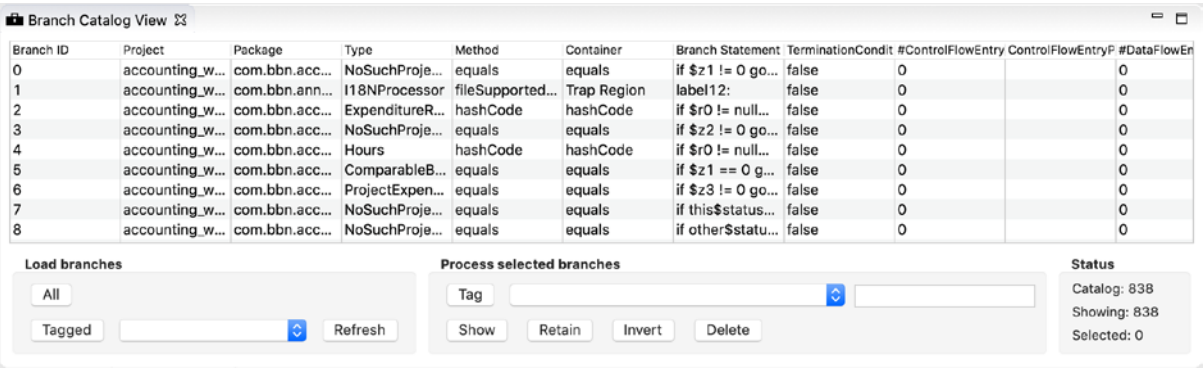

Figure 3. Branch Catalog View

4.2.3 Type and Callsite Catalog

The Type and Callsite Catalog allow analysts to search for, browse, filter, and select types. References and call sites in the app, JDK, or other libraries used by the app. Similar to the loop and Branch Catalog views, analysts can click on a column header to

sort the rows with respect to that characteristic of the type, reference or call site. Types can be selected by specifying the type name, a tag name, or whether they extend. Java's root types in the collections type hierarchy (such as *Iterable* and *Map*). The analyst can show or tag the selected collection types, references and call sites (and optionally, the loops containing or branches governing them) from the user interface. This enables composition of the result with shell queries, Smart Views and subsystem interaction views. This view is useful to narrow down artifacts by enabling analysts to quickly locate collections in the app, loops reachable from references of selected types, and the related call sites. Figure 4 shows a screenshot of the Type and Callsite Catalog view.

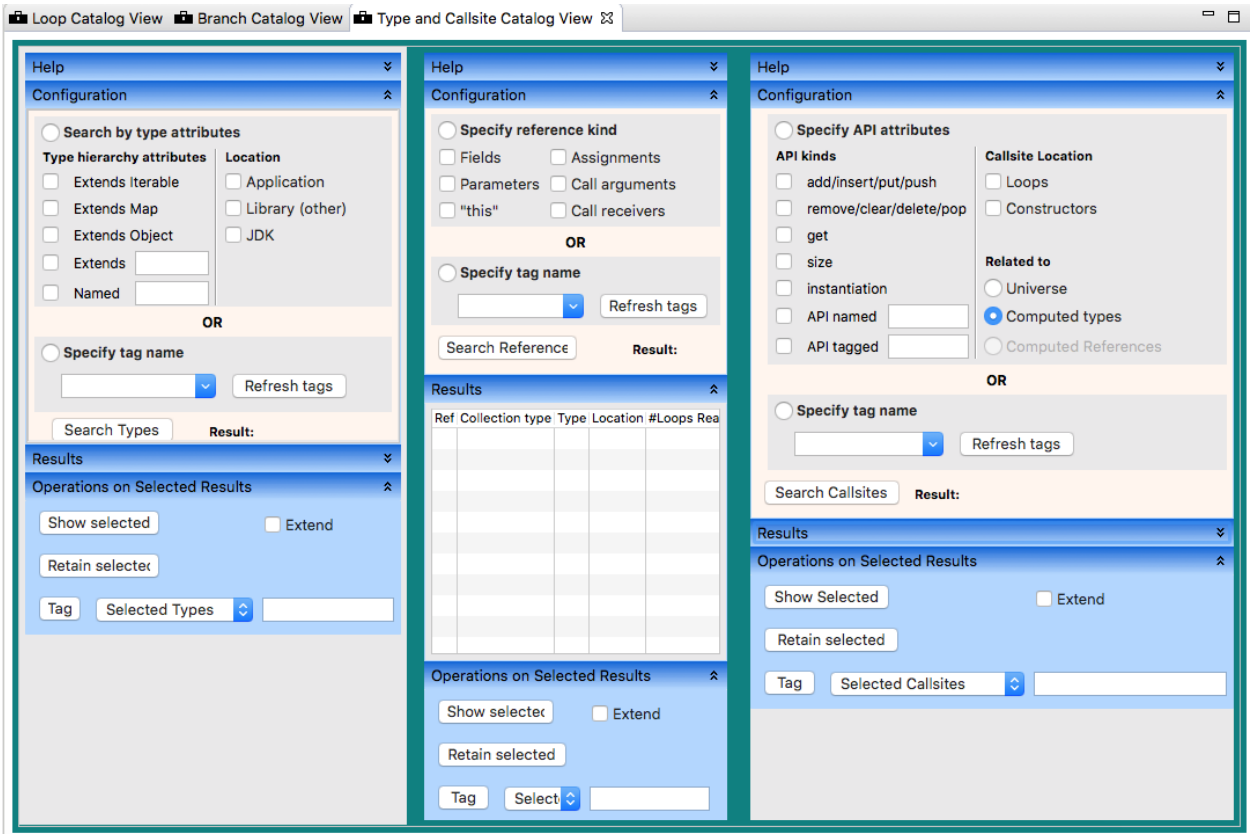

Figure 4. Type and Callsite Catalog View

4.2.4 Lambda Loops Catalog

Java 8 introduced functional programming concepts to Java and added Lambda Expressions to the language. Lambda expressions can be used to form loops, which are implicit loops, e.g. a loop created by a call site such as java.util.stream.Collector.filter(). RULER is capable of detecting such loops called Lambda Loops. The Lambda Loop Catalog tabulates the characteristics of each lambda loop. In total, it lists 15 characteristics of a lambda loop including:

- Source correspondence
- API forming the loop
- Reachability
- Containment relationship with loops and branches

Each column corresponds to a characteristic. Each row corresponds to a lambda expression in the application. Figure 5 shows a screenshot of the Lambda Loop Catalog. To start using the Lambda Loop Catalog, click “All” in the “Load lambda loops” section of the view.

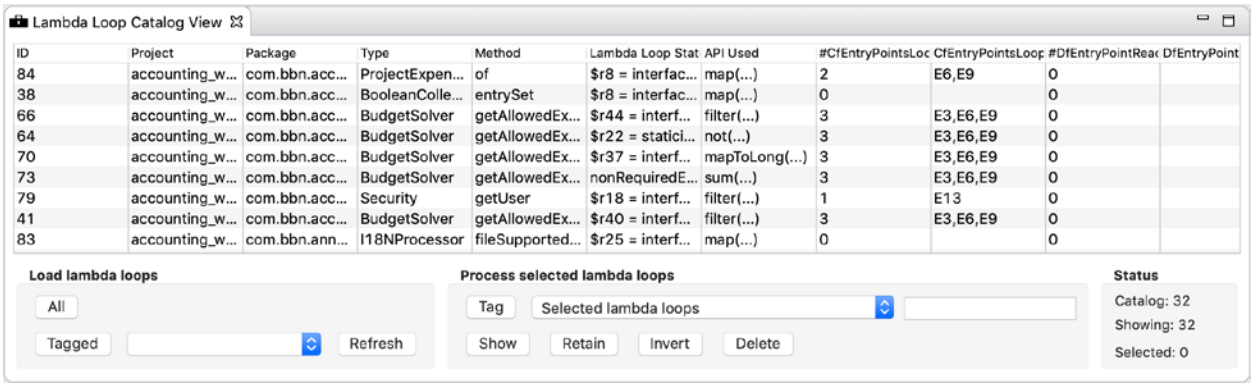

Figure 5. Lambda Loop Catalog

Analysts can click on a column header to sort the lambda loops (rows) with respect to that lambda loop characteristic. Analysts can select a subset of lambda loops[1], which also selects the corresponding lambda expression nodes in the Atlas graph and can be used to drive Smart Views. Selection of a cell under a column representing interaction with a particular subsystem will change the focus of the “Subsystem Interaction” and “Subsystem Interaction PCG” Smart Views to that subsystem.

Lambda loop selections can also be used to control the contents of the view: selected lambda loops may be retained, (soft) deleted, or the selection may be inverted. Analysts can tag selected lambda loops, or load lambda loops with a certain tag. Clicking “All” will reload all lambda loops. This is important to facilitate composition of the lambda loops in the catalog with other analyses and Smart Views.

4.2.3 Loop Call Graph

The Loop Call Graph (LCG) is an induced subgraph of the call graph that retains only methods that contain loops and the call graph between them. The LCG incorporates knowledge of call sites within loops to understand the inter-procedural nesting.

[1] Multi-select is supported on macOS and Windows 10

4.2.4 Projected Control Graph

The number of possible execution paths may be very large, but the number of distinct behaviors along these paths according to criteria of interest may be much smaller. The Projected Control Graph (PCG) is a compact graph derived from the Control Flow Graph (CFG) where CFG paths are grouped into equivalence classes based on behaviors relevant to AC and SC vulnerabilities.

The set of paths in a PCG are in a 1-to-1 correspondence with the equivalence classes of paths in the CFG. The PCG is computed with respect to STAC-specific events that may include: (1) loop headers detected using DLI algorithm [5], (2) call-sites with a call chain to a function containing a loop, and (3) statements that modify loop control variables. Intuitively, the PCG is the subgraph induced by the event nodes and newly added edges wherever necessary to reflect reachability in the graph. Tamrawi et al [6] provide a formal definition of the PCG and presented an efficient algorithm to construct the PCG are documented.

Additional details on PCGs and documentation can be found online at: https://ensoftcorp.github.io/pcg-toolbox.

4.2.5 Loop Termination Dependency Graph

The Loop Termination Dependency Graph (LTD) depicts the data flow dependencies of termination conditions of a given loop. It receives as an input the Control Flow Node marked as a Loop Header and outputs the Termination Dependence Graph (TDG) for the loop. Initially only roots of the graph will be displayed.

4.2.6 Taint Analysis

RULER computes a whole-application version of Taint which is used by downstream automation to call attention to program features which may be reached by externally-controlled input. However, it can be used interactively to follow data dependence anywhere in the application, regardless of whether RULER believes the input is externally controlled. Taint is based on Atlas data flow, with some modifications that improve transitive operations and provide more succinct data dependence graphs.

RULER adds Taint to the XCSG graph primarily by adding tags to existing data flow edges, but also by adding tags to call site edges. This transformation causes calls to library methods to resemble an operator node in data flow. In figure 6 below, the invoked method is CommandLineParser.parse() in the Apache commons library. XCSG normally represents calls sites using all of the edges in the figure, but note that the dashed lines are data flow (XCSG.DataFlow_Edge), whereas the solid lines represent parameters being passed to the call site. The Taint edges are marked in green. Using only data flow, the flow would pass from parameters to invoked method body, and the return value would flow back to the invocation call site. By providing a Taint overlay, RULER can provide a view of the call site, which flows directly from input arguments to method return value.

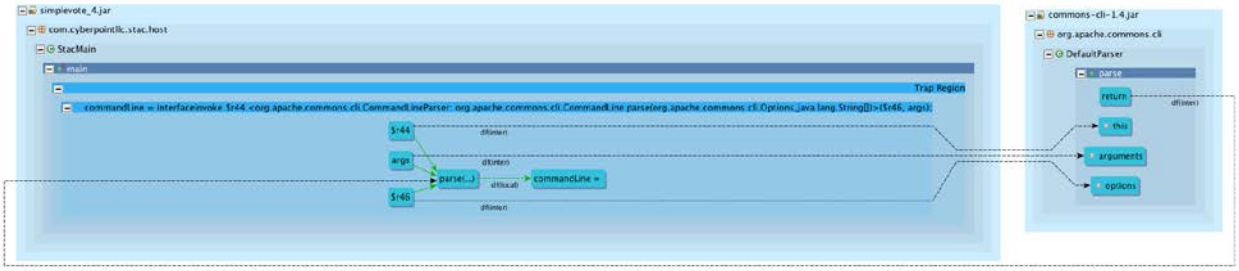

Figure 6. Taint Transformation of CallSite

Figure 7 illustrates a partial Taint graph from a File object, tracing back to the main method's String arguments. The File object represents the password key file. Reading the Taint graph like an AST, it can be observed that the filename is controlled by the -w option. Side note: the implementation of Taint creates cycles as a side effect of ensuring that constructor arguments "Taint" the new object.

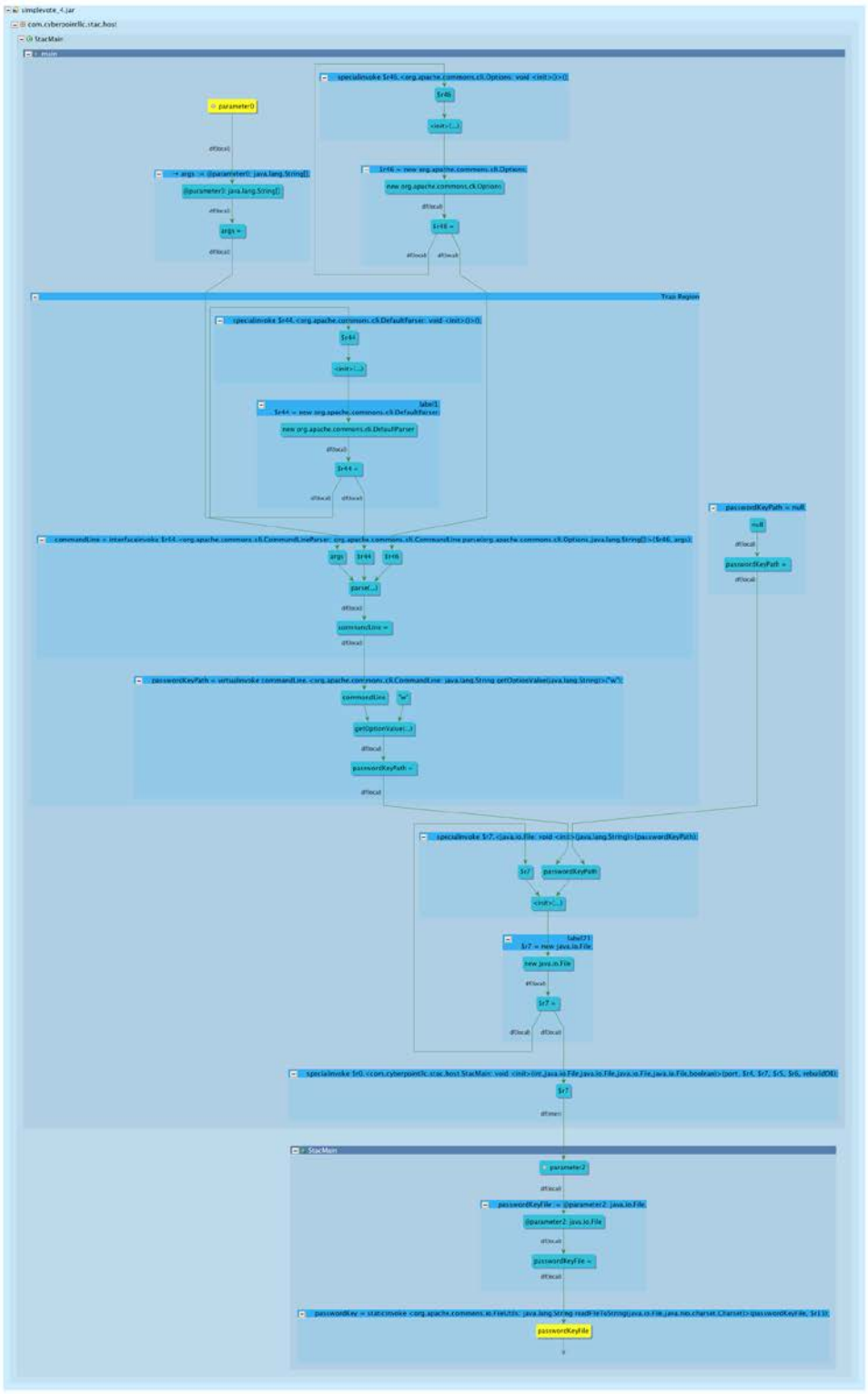

Figure 7. Taint Graph from passwordKeyFile to Main Method

4.2.7 Subsystem Tagging

We classified JDK Application Programming Interfaces (APIs) into functional categories, which we call subsystems of the JDK. The framework for subsystem tagging supports addition of new subsystems in the future. This tagging is useful to investigate the interaction of a method or loop with a particular set of APIs in the JDK. By default, RULER is configured to compute interactions with the Input and Output (IO) subsystem.

The subsystem names, and some key packages/classes whose APIs are included in each subsystem are tabulated in Table 1.

Table 1. JDK Subsystems

Subsystem	**Example packages/classes with APIs belonging to the subsystem**
JAVACORE_SUBSYSTEM	java.util, java.lang
HARDWARE_SUBSYSTEM	javax.sound, javax.sound.midi
IO_SUBSYSTEM	java.nio, java.io
NETWORK_SUBSYSTEM	java.net, javax.net, java.rmi
RMI_SUBSYSTEM	org.omg.CORBA, javax.rmi.CORBA
DATABASE_SUBSYSTEM	javax.sql
LOG_SUBSYSTEM	java.util.logging
SERIALIZATION_SUBSYSTEM	javax.xml.bind, javax.xml.ws.soap
COMPRESSION_SUBSYSTEM	java.util.jar, java.util.zip
UI_SUBSYSTEM	java.applet, java.awt, java.swing

INTROSPECTION_SUBSYSTEM	java.lang.reflect, java.lang.invoke
GARBAGE_COLLECTION_SUBSYSTEM	java.lang.ref
SECURITY_SUBSYSTEM	java.security, javax.security, javax.security.ldap
CRYPTO_SUBSYSTEM	javax.crypto
MATH_SUBSYSTEM	java.math
RANDOM_SUBSYSTEM	java.util.Random, java.security.SecureRandom
THREADING_SUBSYSTEM	java.util.concurrent, java.util.concurrent.atomic
DATA_STRUCTURE_SUBSYSTEM	java.beans
COLLECTION_SUBSYSTEM	java.util.Collection
STREAM_SUBSYSTEM	java.util.stream
ITERATOR_SUBSYSTEM	java.util.Iterator
SPLITERATOR_SUBSYSTEM	java.util.Spliterator
FUNCTIONAL_SUBSYSTEM	java.util.Function

The graph of hierarchy of subsystems is shown in figure 8. An edge in this graph represents a containment relationship. For example, an edge from SPLITERATOR_SUBSYSTEM to ITERATOR_SUBSYSTEM means that SPLITERATOR_SUBSYTEM is contained in ITERATOR_SUBSYSTEM.

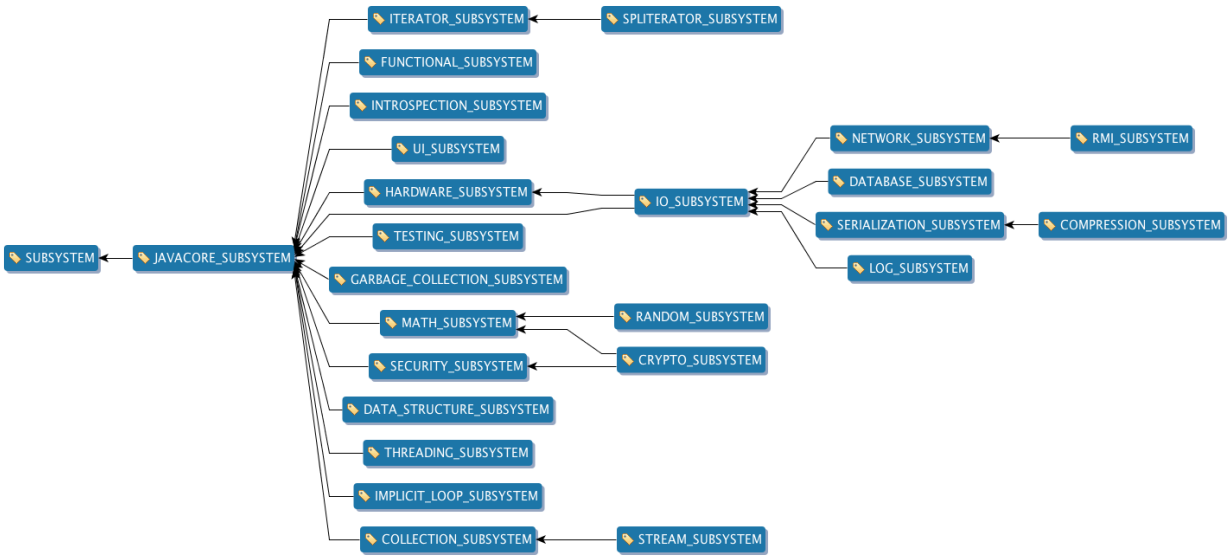

Figure 8. JDK Subsystems Hierarchy

4.3 Gather Corroborating Evidence

4.3.1 Dashboard

The Dashboard is a user interface to view results of canned analyses contributed by RULER. Currently the RULER Dashboard contributes “Properties” and “Smells”. A property is something an analyst should be aware of during an audit (ex: violations of closed world assumptions) whereas a smell is a stronger heuristic that an analyst should prioritize investigating. Whether or not a result is computed and populated in the Dashboard is controlled in the Toolbox Commons (Analyzers) submenu of the Atlas Toolbox preferences menu.

To open the Dashboard view, navigate to *Window -> Show View -> Other… -> Atlas Toolbox -> Dashboard*.

Figure 9 shows the Dashboard view. Initially, it shows only the results of properties/smells which have been computed. To compute results for a new property/smell, an analyst should check the box beside *Search*. This will list all the available properties and smells. Select the property/smell and click on it to expand. Then click *Show All Results* to compute the results and show them.

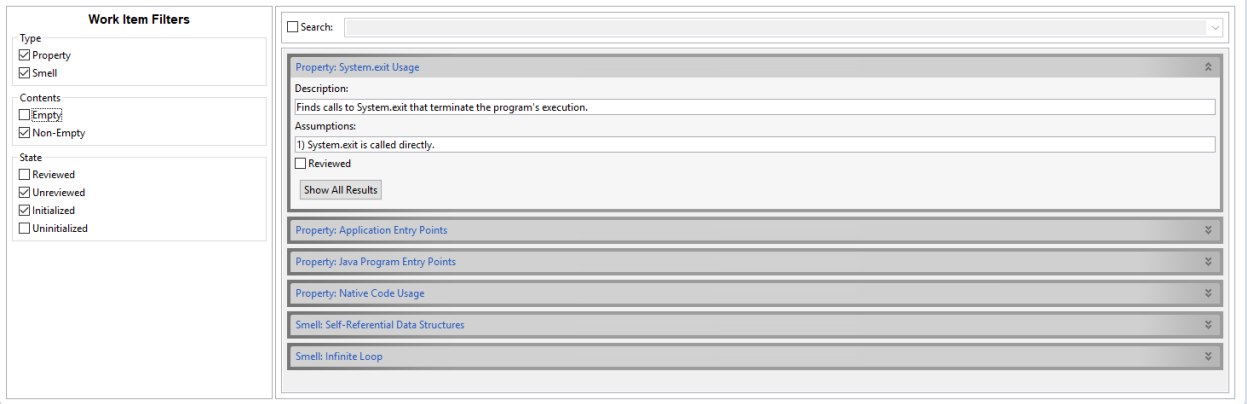

Figure 9. Dashboard Results

The “Properties” and “Smells” provided by the dashboard are as follows.

1. Cyclomatic Complexity

The Dashboard provides the cyclomatic complexity of methods. Using the smell, methods with high cyclomatic complexity are detected.

2. Recursive Functions

The Dashboard provides a smell to call attention to recursive functions in the given app. This smell alerts the analyst about the presence of recursive functions. The analyst can then combine this knowledge with other tools (LCG, PCG etc.) to decide if there is a vulnerability.

3. System.exit Usage

The Dashboard can find methods with System.exit usage. System.exit() causes the app to terminate execution and stop all threads. This can leave a client instance waiting forever and calls for attention of the analyst. This smell exactly does that.

4. Self-Referential Data Structures

The Dashboard provides a smell to find self-referential data structures in the given app. A self-referential data structure has fields of the same type and can grow very quickly when used in a loop. This also increases the cost (space and time) of processing the information stored in such data structures. In practice, we observed these causing vulnerabilities, especially AC vulnerabilities. Hence, the analyst should be alerted about the presence of self-referential data structures which is what this smell does.

5. Deeply Nested Loops

The Dashboard provides a smell to find loops with (local) nesting depth ≥ 4. The nesting depth is also provided in the Loop Catalog view for all loops.

6. Infinite Loop

The Dashboard provides a smell to find certain classes of infinite loops (result of termination condition is constant e.g. while(true)). Presence of infinite loops is an obvious indication of a potential AC vulnerability. Thus, the analyst should be alerted of their presence. An analyst can also find them using the Loop Catalog. Since this is an often used smell, it is provided as a canned analysis.

7. Executors.newFixedThreadPool Usage

The Dashboard provides a smell to find calls to Executors.newFixedThreadPool. Fixed size thread pools are of interest for AC vulnerabilities. If all threads can be kept busy, it can lead to blocking a benign request.

8. Order-Preserving Data Structures

The JDK provides various data structures, which can be used to store data while maintaining an ordering, e.g. Lists, TreeSets, etc. The Dashboard finds such data structures in the given app. While such data structures do not cause vulnerabilities by themselves, we observed vulnerabilities that depend upon the order in which elements stored in the data structures are processed. For such vulnerabilities, presence of order-preserving data structures is a necessary condition. This smell brings such vulnerabilities under the radar of the analyst and they can then confirm the presence of this class of vulnerabilities by analyzing the code that processes these data structures using tools such as the IPCG (PCG Builder).

Note: Arrays are not considered.

9. Reflection Usage

The Dashboard provides a smell to find all Java Reflection usage via java.util.reflection in the app. This alerts the analyst to hidden behaviors, as reflection can be used to obfuscate method invocation and data transfer.

10. Random Usage in Loop Body

The Dashboard provides a smell to find loops, which reference the Random API. This was developed in response to STAC challenge applications where loop iteration bounds were obfuscated using random. For example, the following loop computes the same result as it would if the inner loop conditions were omitted, but has a slightly increased runtime because a quarter of the inner loops will fail the second condition. This pattern interferes with classifying the loop bounds as monotonic, and increases the apparent loop nesting. Therefore, this smell serves a different purpose, as it does not detect vulnerable loops: it helps to find loops, which are not vulnerable but are likely to raise false alarms. Such loops should be filtered out from further analysis and this smell helps to do so.

```
      while (i < length) {
            while (i < length && random() < 0.75) {
                  i++;
            }
      }
```

11. Application Entry Points

The Dashboard finds all the methods, which serve as entry points to the given app. Entry points typically serve as the points of interactions between the attacker and the app. Thus, it is common to begin the analysis at the entry points. This property is computed to find the application entry points for the analyst.

Since the apps typically use various web frameworks, the web framework handlers must also be considered as entry points. RULER uses heuristics to recognize the handlers of the web frameworks. In particular, RULER recognizes the following as application entry points:

- java main methods in the app
- javax.servlet.http.HttpServletRequest (Java servlet requests)
- com.sun.net.httpserver.HttpHandler (Sun Http request handlers)
- Netty channelRead0() method (Netty framework)
- Methods annotated with org.springframework.web.bind.annotation (Spring framework)
- Methods that override spark.Route() (Spark framework)
- accept() methods in java.net.Socket() (Socket connections)

12. Class Loader Usage

The Dashboard finds calls to Class Loaders, for reasons similar to reflection usage - it can be involved in obfuscation of class usage.

13. Native Code Usage

The Dashboard can find all the uses of native methods in the app. This may indicate hidden behaviors, as native code can potentially do anything, and is not otherwise visible to a purely bytecode-based analysis.

14. Java Program Entry Points

The Dashboard can find all Java main methods in the given app.

15. Serialization Usage

Dashboard can find all uses of the Serializable interface. In practice, we observed improper handling of serializable objects causes vulnerabilities. E.g. XML bomb is a well-known attack that occurs when an app tries to serialize a malicious XML document. This smell alerts the analyst of the presence of serializable objects in the app. This brings the known attacks in scope and they should be tested first.

16. Process Usage

The Dashboard can find calls to all the methods that allow the application to run a shell command, as this may represent hidden behavior of interest to an analyst.

4.3.2 PCG Builder

The PCG Builder interface is a utility for selecting events across multiple functions and computing an interprocedural PCG (IPCG). For basic use, simply select the events of interest and press the “Show PCG” button. For advanced use, an analyst may select ancestor functions and expandable functions for additional context.

To open the PCG Builder view, navigate to *Window -> Show View -> Other… -> Atlas Toolbox -> PCG Builder*.

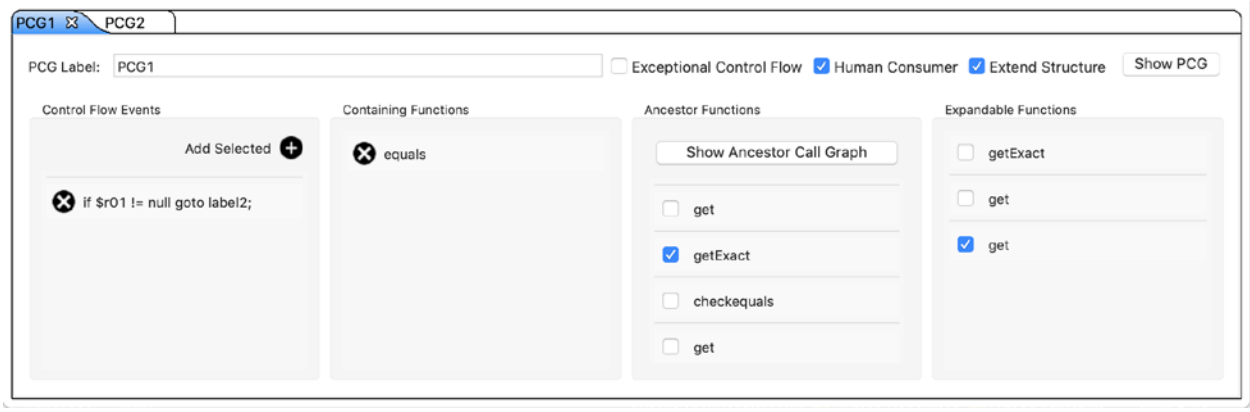

Figure 10. PCG Builder

Exceptional Control Flow denotes the control flow paths due to exception handling. Select this option to include exceptional control flow paths in the computation of PCG.

Human Consumer: When an IPCG is created, the call sites of the functions are included as events. As per the definition of control flow paths, the return nodes of the functions should be redirected back to the function containing the call sites. This though creates unnecessary information that puts cognitive burden on the human analyst. To reduce this we provide the option to not include the redirection edges and make IPCG more suitable for a human consumer.

Extend Structure: The PCGs created contain the control flow nodes and edges that form the relevant control flow paths corresponding to the selected events. These events are contained in various functions and the functions themselves are contained in different Classes, the Classes are contained in different packages and so on. This is the extended structure of the PCG. The extended structure need not be shown depending upon the analyst's preference. This option enables/disables extension of the structure.

Ancestor functions are functions that exist in the reverse call graph of any functions containing selected events. Selecting ancestor functions increases the size of the IPCG, but adds additional calling context to the result.

Expandable functions add call sites within the ancestor functions along the path to the events. Once ancestor functions are selected, functions may be expanded using the checkboxes under "Expandable Functions". Call sites to functions that lead to events are implicit events in an IPCG. By default, the implicit events are elided in functions that do not contain explicit events and the entire function is collapsed to a single node in the resulting PCG. However, these collapsed functions can be expanded by the PCG Builder interface to reveal PCGs of implicit events for each expanded node.

The PCG Builder also contains a new PCG instance button on the menu bar, which creates a new tab. This way the PCG builder can maintain the state of several instances of IPCGs.

4.3.3 Loop Subsystem Interactions Builder

The Loop Subsystem Interaction Builder allows an analyst to specify a set of loops and a set of functions as input, and select only the loops that have a call interaction with at least one of the input functions. For example, to select a set of loops that interact with collection APIs, an analyst can first display, select the loops to be filtered, and add them to the left pane "Loops" (see Figure below). To add the functions, the analyst can run the shell command to first display all the collection subsystem APIs, and then add all (or a subset of) the displayed nodes to the right pane "Functions" in the builder. Clicking on "Show Loops Interacting with Subsystem" displays the loops that interact with the selected loops via the call graph.

```
show(nodes("COLLECTION_SUBSYSTEM").contained.nodes(XCSG.Method))
```

To open the Loop Subsystem Interaction Builder, navigate to *Window -> Show View -> Other… -> Atlas Toolbox -> Loop Subsystem Interaction Builder*.

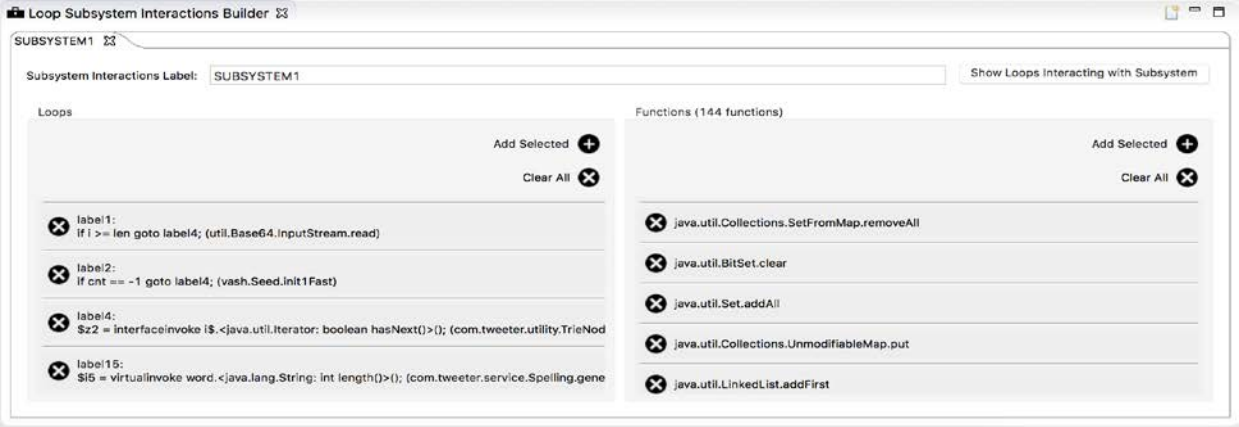

Figure 11. Loop Subsystems Interactions Builder

4.3.4 Decompiler view

The Decompiler view provides two-way correspondence between Jimple code and decompiled source code to enhance the comprehension of bytecode. By selecting a portion of a Jimple code or Atlas graph element corresponding to a method, the Decompiler view will output the decompiled source code corresponding to the selected Jimple code or Atlas graph element.

The view uses the Class File Reader (CFR) tool to decompile Java bytecode. The view works on a per-method basis, i.e., when an analyst selects a method, it decompiles only the source code for that method on the fly. Figure 12 shows a screenshot of the CFR Decompiler View

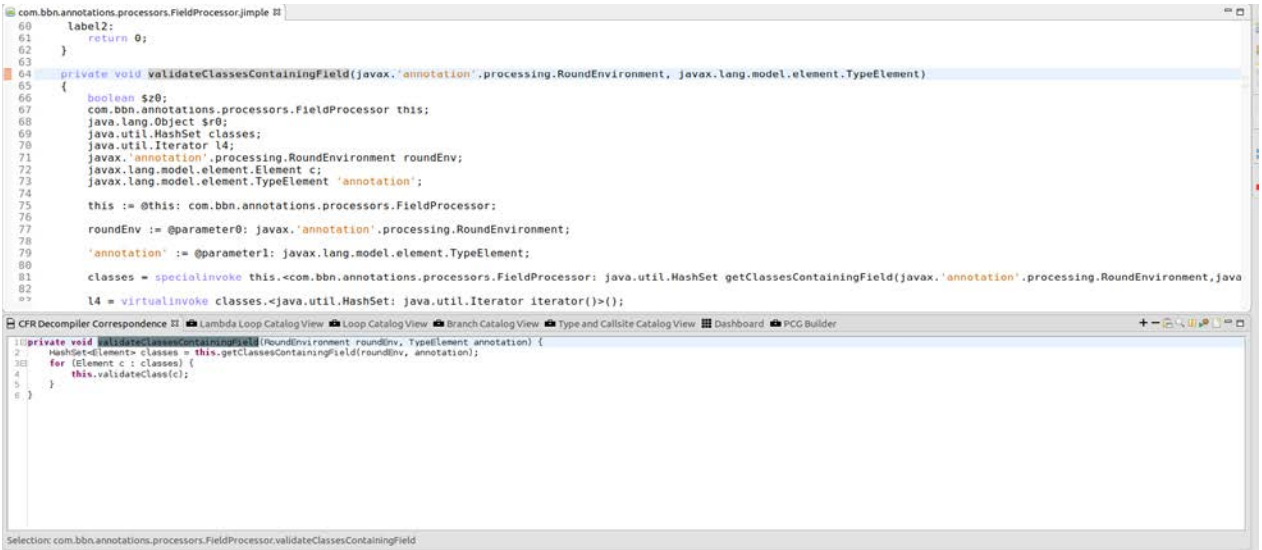

Figure 12. CFR Decompiler View

4.3.5 Filter View

The Filter View allows an analyst to specify a “root set” of nodes and edges as input and then selectively apply filters to the set to produce a subset of the original input. The interface loads RULER-contributed filters and automatically determines which filters are applicable to the given input sets.

To open the Filter View, navigate to *Window -> Show View -> Other… -> Atlas Toolbox -> Filter View.*

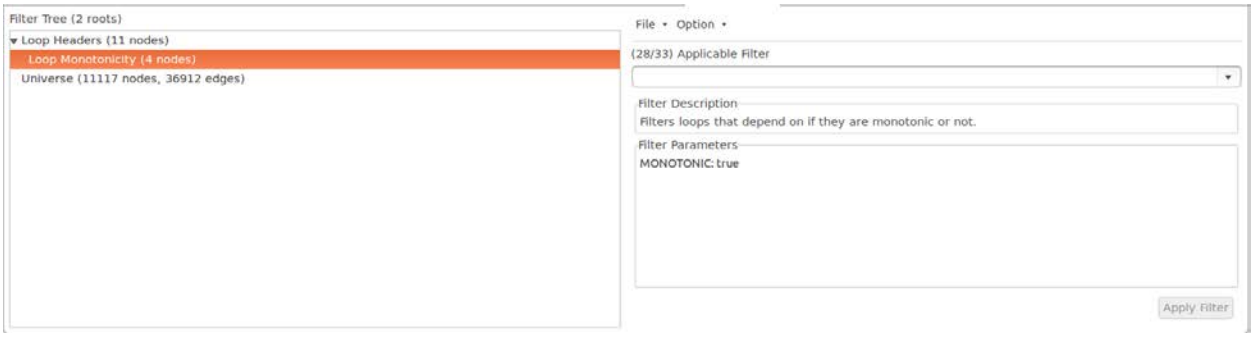

Figure 13. Filter View

Additional details can be found online at: https://ensoftcorp.github.io/toolbox-commons.

4.3.6 Smart Views

RULER currently contributes the Smart Views listed in this section (by title). To open a Smart View, use the menu: *Atlas -> Open Atlas Smart View*. Then select a specific view using the combo box on the bottom of the Smart View. Smart Views respond to selections from other graphs or from the text-based Jimple editors. The main contributed Smart Views include:

- **Code Painter:** The Code Painter interface and Control Panel are an advanced Smart View implementation that allows additional control over Smart View behaviors and styling information. Two new concepts are introduced: 1) a Code Painter and Code Painter Control Panel and 2) a Color Palette. The Code Painter interface is an extensible interface for defining new Code Painter Smart Views. The Smart Views are controlled by selections and the Code Painter Control Panel. Color Palettes are styling logic that can be applied individually or as a layer in the active styling of the Code Painter view.
- **Enhanced Loop Call Graph:** Overview of loops in the program, in terms of loop headers and call sites. Edges in an Enhanced Loop Call Graph link 1) from CallSites to possible target Methods, and 2) from Methods to Loop and CallSites in the body. Edges represent nesting levels. For example, a CallSite in a Loop will be reachable along a path: Method > Loop > CallSite. See also: LCG.

- **Loop Call Graph (LCG):** Overview of loops in the program, in terms of the call graph. See also: LCG.
- **Control Flow Graph (CFG):** Overview of Control Flow within a Method.
- **Control Flow Graph with Exceptional CF (Ex CFG):** Overview of Control Flow and Exceptional Control Flow within a Method.
- **Projected Control Graph (PCG):** Create a PCG with respect to events. See also: PCG.
- **PCG: Events = {Loop Headers, CallSites to LCG}:** Similar to the view Projected Control Graph (PCG), with the automatic inclusion of events based on the Loop Call Graph (LCG).
- **Taint:** View dependencies based on Taint edges. RULER currently defines Taint edges as 1) Local Data Flow, and 2) edges from actual arguments to CallSites, for CallSites, which resolve to the JDK library (which has the effect of treating the CallSites as Operators). See also: Taint.
- **Taint -> PCG:** Given a subgraph based on Taint, create a PCG. See also: Projected Control Graph (PCG), Taint.
- **PCG + Condition Flows:** Include data dependencies of conditions for a PCG. See also: Projected Control Graph (PCG), Taint.
- **Loop Terminating Conditions (LTCs):** see LTC.
- **Loop Termination PCG:** To create a Projected Control Graph that helps to quickly understand the structure of a loop. See also: Projected Control Graph (PCG)
- **Loop Termination Dependence Graph (TDG):** Show data flow dependencies of termination conditions of a given loop.
- **Subsystem Interaction View:** To help understand the interactions a loop, a method, a control flow node or a call site has with a subset of the pre-defined subsystems. See also: Subsystem Interaction Tagging**.**
- **Subsystem Interaction PCG View:** To help understand the interactions a loop or a method has with the rest of the app or one of the pre-defined functional categories of APIs in the JDK (subsystems). See also: Subsystem Interaction Tagging.
- **Loop Nesting View:** To understand the nesting relationships between loops in a selected method, or loops contained within a selected loop header.
- **Reachable Loops View (From/To Input):** To identify loops reachable from/to a selected application input or variable.
- **Loop Body View:** To understand the control flow within a selected loop.

4.4 Targeted Dynamic Analysis to Generate Exploits

We developed the Mockingbird framework [2] that combines static and dynamic analyses to yield an efficient and scalable approach to analyze large Java software. The framework is an innovative integration of existing static and dynamic analysis tools and a newly developed component called the Object Mocker that enables the integration. The static analyzers are used to extract potentially vulnerable parts from large software.

Targeted dynamic analysis is used to analyze just the potentially vulnerable parts to check whether the vulnerability can actually be exploited.

Mockingbird framework incorporates:

- The multitude of static analyzers built using the Atlas platform. These analyzers are used to extract the relevant code - the code that is potentially vulnerable.
- The AFL fuzzer supplemented by the Kelinci, a tool that interfaces AFL to operate on Java programs.
- The Object Mocker that can automatically create harnesses to apply the AFL fuzzer to dynamically analyze the relevant code extracted using the static analyzers.

The key novelty of the Mockingbird is targeted dynamic analysis (TDA). TDA performs dynamic analysis of just the relevant code using binary inputs. The framework provides the capability to mock any Java object type and produces a testing harness that transforms binary inputs to appropriate object types. For fuzzers such as AFL, users manually create a test harness that translates the binary inputs from the fuzzer to the data structures expected by the target program. Developing the harnesses manually for just the relevant code is difficult and laborious because the program state must be considered as the input. The program state is communicated to the relevant function via the stack memory using function parameters and return values or through the heap memory, using reads and writes to global variables. The harness incorporates mocked objects that mimic the program artifacts that carry the inputs into the relevant code. The Mockingbird framework creates the harnesses automatically. The framework earns its name from the term “mock objects” as used by the testing community.

The Mockingbird framework optionally allows constraints to be placed on the values of object fields that store the encapsulated program data. With this enhancement, it is possible to systematically incorporate domain-specific knowledge into automatically generated test harnesses. For example, if a program only operates on byte arrays that begin with a magic sequence, such as 0xFFD8 in the case of JPEG file formats, then the test harness can simply prefix the program input with the known sequence to increase the chances of quickly driving the program execution in meaningful ways. The Mockingbird framework supports configurable input generation constraints to be specified for test harnesses as a means for injecting domain knowledge.

An Illustrative Example: We examine a DARPA STAC challenge application called BraidIt for AC vulnerabilities. The application’s source code is available at: https://github.com/Apogee-Research/STAC. The case study demonstrates: (1) the use of Mockingbird framework to efficiently discover an algorithmic complexity vulnerability, and (2) a follow-up experiment using Mockingbird’s APIs to create a custom tool for computing program invariants that provide holistic understanding of the vulnerability. The follow-up experiment reveals not just one input but a class of inputs that cause the vulnerability; in particular, it reveals the smallest input that can cause the vulnerability.

5. Analyzing Challenge Applications

It was observed that analysts follow certain patterns in the workflow to find each type of Algorithmic Complexity (AC) or Side Channel (SC) vulnerability (AC Time, AC Space, SC Time, and SC Space). These patterns were distilled and the workflow of analysts was documented. Here are the steps taken for majority of the challenge questions:

5.1 AC Time

1. Find App entry points using Dashboard (Entry points).
2. Find a mapping between the app entry points and corresponding request handlers using DataFlow/Taint Analysis toolbox with the help of PCG toolbox for compact graphs.
3. Find user-input controlled requests that the attacker can leverage using DataFlow/Taint Analysis toolbox.
4. Find user-input controlled loops reachable from the user-input controlled requests using Loop Catalog, Loop Hierarchy Smart view, and Dataflow/Taint Analysis toolbox.
5. Find user-input controlled loops that can inflate recursive data structure using Type and Callsite Catalog and Loop Catalog.
6. Prioritize loops auditing based on domain knowledge, nesting depth, recursion, complexity of terminating conditions, and interaction with Java subsystems using Loop Catalog, Loop Hierarchy Smart view, and Dataflow/Taint Analysis toolbox.
7. Verify the existence of AC time via Dynamic Analysis.

5.2 AC Space

1. Find specific operations or types of interest based on the challenge question using Type/Callsite Catalog along with Subsystem Interaction View.
2. Find App entry points using Dashboard (Entry points).
3. Answer, “Are the operations or types of interest reachable from entry points (i.e., user-input controlled)?” using Dataflow/Taint Analysis toolbox.
4. Answer, “Are these operations or types of interest governed by loops or interesting branches that are user-controlled?” using Loop Catalog.
5. Answer, “What things are being consumed by the operations or types of interest?” using DataFlow/Taint Analysis.
6. Answer, “Are the things being consumed user-controlled? Can they be inflated or occurs within a loop?” using DataFlow/Taint Analysis.
7. Understand the inter-procedural relationship and call structure leading to the user-controlled objects that will be consumed using IPCG and Call Graph.
8. Verify if the inflation in space can be done dynamically to exceed the budget using Dynamic Analysis.

5.3 SC Time

1. Find objects corresponding to the secret.
2. Find App entry points using Dashboard.

3. Find a mapping between the entry points and corresponding request handlers using DataFlow/Taint Analysis with the help of PCG.
4. Find a mapping between requests and possible responses using DataFlow/Taint Analysis with the help of PCG.
5. For each mapping of a request to responses, find if there is a user-controlled loop/recursion, intensive operation, or throwing of unhandled exceptions using Loop/Branch Catalog, Type and CallSite View with the help of PCG.
6. Prioritize audit of request/response mapping based on their affect/relation on revealing the secret.
7. Check if the differential time can be observed to reveal the secret using Dynamic Analysis.

5.4 SC Space

1. Find objects corresponding to the secret using DataFlow/Taint Analysis.
2. Find App entry points using Dashboard.
3. Find a mapping between requests and request handlers using DataFlow/Taint Analysis along with PCG.
4. Find app output points using Subsystem Interaction View along with Type and Callsite Catalog.
5. Determine if output points are, user-input controlled and has a relation to the secret using DataFlow/Taint Analysis.
6. Study what are the possible ways that an attacker can provide input to inflate theses output controlled objects.
7. Determine if user-input controlled output points are governed by loops and differential conditions using Loop Catalog along with IPCG and system dependency graph view.
8. Check if the differential space can be observed to reveal the secret using Dynamic Analysis.

6 Results and Discussion

6.1 Results

All blue teams participated in seven engagements throughout the STAC program.

- Engagement 1 (E1) was a live engagement that took place in February 2016 at DARPA.
- Engagement 2 (E2) was a take home version of Engagement 1 which ended in July 2016.
- Engagement 3 (E3) was a live engagement that took place in January 2017.
- Engagement 4 (E4) was a take home version of Engagement 3 which ended in June 2017.
- Engagement 5 (E5) had a home component that ended in January 2018 and a live collaborative engagement at DARPA in February 2018.

- Engagement 6 (E6) had a home component and a live collaborative engagement at DARPA in August 2018.
- Engagement 7 (E7) had a home component and a live collaborative engagement at DARPA in February 2019.

A summary of the ISU-EnSoft team results is shown in the figure below. Results are shown for the end engagement for each set of challenge applications. For example, E1 and E2 were based on the same challenge applications; and so were E3 and E4. E5, E6 and E7 each had a home component, followed by the live one and the results after the live engagement are shown below.

6.2 Engagement Observations

Figure 14 shows that our accuracy continued to increase for each engagement (except the last one). Up to the last engagement E7, teams could choose to analyze only a subset of the applications questions. (The attempted column reflects that). As of E7, all teams were required to answer all the questions. That is the reason the accuracy has dipped a little for E7.

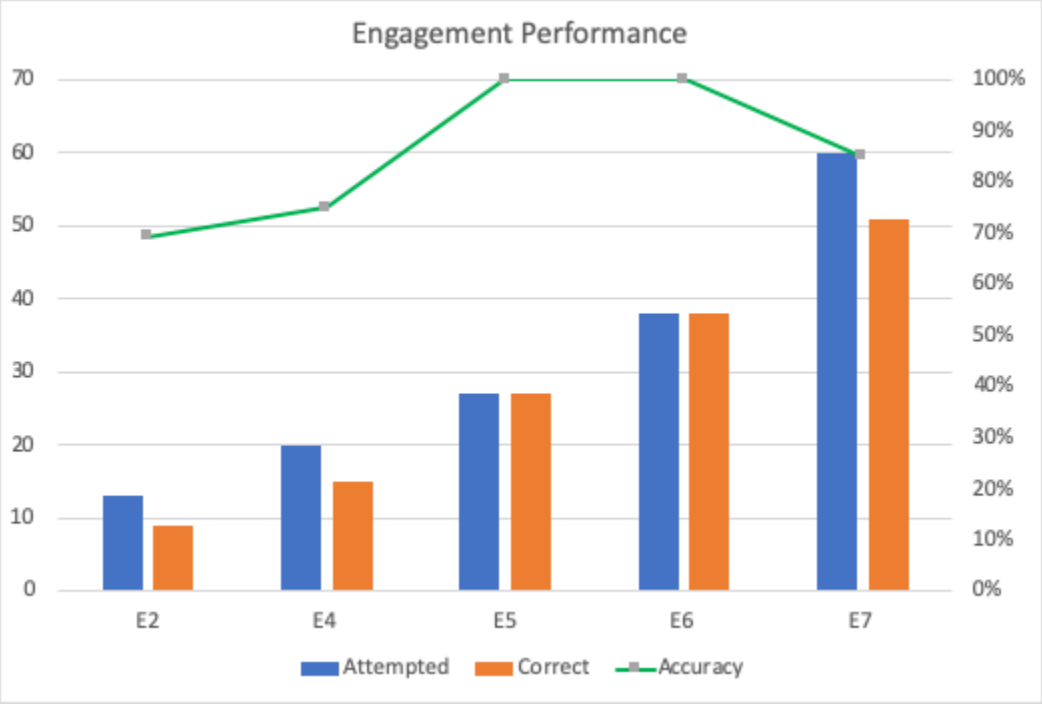

Figure 14. Engagement performance.

7. Conclusions

The STAC program was quite challenging. While the program was restricted to algorithmic complexity (AC) and side channel (SC) as the two classes of vulnerabilities, they involve almost endless variations of triggers and malicious payloads. Coming up with analyses to detect these vulnerabilities is extremely challenging because of these variations. It often entails complex analyses involving loops. Existing techniques from the large body of research on loop analyses were not particularly useful for addressing problems from the engagements, and a new approach was needed.

The STAC program enabled significant advances. The foremost advance is the intelligence amplifying (IA) technology for detecting AC and SC vulnerabilities. Fredrick Brooks, recipient of Turing Award and the US National Medal of Technology, writes “If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.” Complex software engineering problems cannot be solved by AI, it calls for IA technology. An IA automation technology for addressing complex software problems is presented in the report.

Several challenges were encountered in coming up with an effective way to combine automated analysis with human ingenuity. This report discussed these challenges and how to address them. The advances in the IA technology lead to steady improvement in the performance on the engagements. The program manager recommended the IA technology for potential adoption.

The take-home and on-site engagements very useful to drive the research. The engagements enabled identification of the weaknesses in the approach and provided concrete directions to refine it. The collaborative part added to the engagements was especially useful. Understanding the approaches taken by other teams was helpful and all teams benefitted by learning from each other.

8. Publications and Presentations

8.1 Journal Papers

1. Ahmed Tamrawi and Suresh Kothari, “Projected Control Graph for Computing Relevant Program Behaviors”**,** Science of Computer Programming Volume 163, Pages 93-114, October 2018.

8.2 Book Chapter

1. Suresh Kothari, Ganesh Ram Santhanam, Payas Awadhutkar, Benjamin Holland, Jon Mathews, and Ahmed Tamrawi, “Catastrophic Cyber Physical Malware,” Springer Verlag Publishers, October 2018, Pages 1-41.

8.3 Peer-Reviewed Conference Papers

1. Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, “DISCOVER: Detecting Algorithmic Complexity Vulnerabilities,” 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).
2. Derrick Lockwood, Benjamin Holland, Suresh Kothari, “MOCKINGBIRD: A Framework for Enabling Targeted Dynamic Analysis of Java Programs,” 41st International Conference on Software Engineering (ICSE 2018), Montreal, May 2019.
3. Ahmed Tamrawi, Sharwan Ram, Payas Awadhutkar, Benjamin Holland, Ganesh Ram Santhanam, Suresh Kothari, “DySDoc3 – DynaDoc: Automated On-Demand Context-Specific Documentation,” 2018 DOCGEN Challenge Workshop at the IEEE International Conference on Software Maintenance Evolution (ICSME 2018), Madrid, Spain, September 25, 2018, (***Best Tool Award***).
4. Suresh Kothari, “Demystifying Cyber-Physical Malware,” 40th International Conference on Software Engineering (ICSE 2018), Technical Briefing, Sweden, Gothenburg, May 27-June 3 2018.
5. Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, Jon Mathews, “COMB: Computing Relevant Program Behaviors,” 40th International Conference on Software Engineering (ICSE 2018), Sweden, Gothenburg, May 27-June 3 2018.
6. Ganesh Ram Santhanam, Ben Holland, Suresh Kothari, Nikhil Ranade, “Human-on- the-loop-automation for Detecting Software Side Channel Vulnerabilities,” 13th International Information Systems Security Conference (ICISS 2017), Bombay I.I.T., Springer LNCS 10717, pp. 209-230, December 2017.
7. Payas Awadhutkar, Ganesh Ram Santhanam, Ben Holland, Suresh Kothari, “Intelligence Amplifying Loop Complexity Characterizations for Detecting Algorithmic Complexity Vulnerabilities,” the 24th Asia-Pacific Software Engineering Conference (APSEC), December 2017.
8. Suresh Kothari, Payas Awadhutkar, Ahmed Tamrawi, Jon Mathews, “Modeling Lessons from Verifying Large Software Systems for Safety and Security,” Proceedings of the 2017 Winter Simulation Conference, November 2017.
9. Ganesh Ram Santhanam*, Ben Holland, Jon Mathews, Suresh Kothari, “Interactive Visualization Toolbox to Detect Sophisticated Android Malware,” the 14th IEEE Symposium on Visualization for Cyber Security (VizSec), October 2017.
10. Ben Holland, Ganesh Ram Santhanam, Suresh Kothari, “Transferring state-of-the-art immutability analyses: An experimentation toolbox and accuracy benchmark,” IEEE International Conference on Software Testing, Verification and Validation, Tokyo, Japan, March 2017.
11. Ahmed Tamrawi, Suresh Kothari, “APSEC 2016 – Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities,” 23rd Asia-Pacific Software Engineering Conference, Hamilton, New Zealand, December 6-9, 2016. (***Invited for journal publication***)
12. Benjamin Holland+, Ganesh Santhanam*, Payas Awadhutkar+, Suresh Kothari, “Statically-informed Dynamic Analysis Tools to Detect Algorithmic Complexity

Vulnerabilities," IEEE Source Code Analysis and Manipulation (SCAM) Conference, Raleigh, North Carolina, October 1-3, 2016.
13. Ahmed Tamrawi, Payas Awadhutkar, Suresh Kothari, "Insights for Practicing Engineers from a Formal Verification Study of the Linux Kernel," Formal Verification for Practicing Engineers Workshop, IEEE ISSRE 2016, Ottawa, Canada, October 23-27, 2016 **(*Invited for journal publication*)**.
14. Suresh Kothari, Ahmed Tamrawi, and Jon Mathews, "Human-Machine Resolution of Invisible Control Flow," IEEE International Conference on Program Comprehension (ICPC 2016), pp. 226-229, May 15-16, Austin, Texas, May 15-16, 2016.
15. Suresh Kothari, Ahmed Tamrawi, and Jon Mathews, "Refactoring Verification: Accuracy, Efficiency and Scalability through Human-Machine Collaboration," A Technical Briefing Paper, IEEE International Conference on Software Engineering (ICSE 2016), Austin, Texas, May 15-19, 2016.
16. Suresh Kothari, Ahmed Tamrawi, Jeremias Sauceda, and Jon Mathews, "Let's Verify Linux: Accelerated Learning of Analytical Reasoning through Automation and Collaboration," a Software Engineering Education Track (SEET) paper, IEEE International Conference on Software Engineering (ICSE 2016), Austin, Texas, May 15-19, 2016.
17. Suresh Kothari, "Software Engineering Research Lab to Airplanes, Orion and Beyond," a Software Engineering Research and Industry Practice (SER&IP) Workshop paper, IEEE International Conference on Software Engineering (ICSE 2016), Austin, Texas, May 15-19, 2016 (***Best Paper Award***).

8.4 Keynotes and Invited Talks given by Suresh Kothari

1. "An 18th-century Mathematician, a $336 Million Patent, and Software Verifiability," CyLab Distinguished Seminar Series at Carnegie Mellon University, April 2019.
2. "An 18th-century Mathematician, a $336 Million Patent, and Software. Experimentation," International Conference on Security and Privacy, Jaipur, India, January 11, 2019 ***(Keynote Talk)***.
3. "An 18th-century Mathematician, a $336 Million Patent, and Software Experimentation," Computer Science Department, University of British Columbia, October 2019.
4. "Software Security Headaches: Analgesic or Hospital," Security of Information and Networks (SIN 2017), Manipal University, Jaipur, India, October 14, 2017 ***(Keynote talk).***
5. "Reflections on Applying Mathematics to Solve Hard Software Problems," University of Central Florida, August 29, 2017.
6. "Demystifying Cybersecurity for CPS Community," MathWorks Research Summit, Boston, June 4, 2017.
7. "Euler, the 336 Million Dollar Software Patent: Reflecting on How to Solve Hard Software Problems," ACSS2017, Patna, India, March 19, 2017.
8. "Euler, the 336 Million Dollar Software Patent: Reflecting on How to Solve Hard Software Problems," University of Auckland, Auckland, New Zealand, December 2, 2016.

9. “Euler, the 336 Million Dollar Software Patent: Reflecting on How to Solve Hard Software Problems,” Auckland University of Technology, Auckland, New Zealand, December 1, 2016.
10. “Euler, the 336 Million Dollar Software Patent: Reflecting on How to Solve Hard Software Problems,” Fraunhofer Institute, Baltimore, November 3, 2016.
11. Insights for Practicing Engineers from a Formal Verification Study of the Linux Kernel,” Plenary Talk, Formal Verification for Practicing Engineers Workshop at ISSRE 2016, October 5, 2016 (***Plenary Talk***).
12. Technical Briefing, International Conference on Software Engineering, Austin, May 17, 2016.
13. “Rethinking Verification: Accuracy, Efficiency, and Scalability through Human-Machine Collaboration,” Georgia Institute of Technology, Atlanta, May 12, 2016.
14. “Rethinking Malware Detection: Accuracy, Efficiency, and Scalability through Human- Machine Collaboration,” University of Maryland, Baltimore, February 17, 2016.
15. “Program Analysis and Reasoning for Hard-to-Detect Software Vulnerabilities,” Jadavpur University, Kolkata, India, December 17, 2015.
16. “Modeling Critical Software Systems for Cybersecurity and Safety,” Tata Research Design and Development Center (TRDDC), Pune, India, December 7, 2015.
17. “Evidence-Enabled Collaborative Verification for Cybersecurity and Safety of Critical Software Systems,” Government of India Center for Development of Advanced Computing (CDAC), Pune, November 24, 2015.
18. “Visual Models to Solve Hard Problems at the Intersection of Cybersecurity and Software Reliability,” Rockwell Collins Headquarters, Cedar Rapids, Iowa, October 22, 2015.

8.5 Tutorials and Short Courses

1. Suresh Kothari, “Software Engineering and Cybersecurity,” short course at Indian Institute of Technology, Jammu, India, January 7-9, 2019.
2. Suresh Kothari and Ben Holland, “Systematic Exploration of Critical Software for Catastrophic Cyber-Physical Malware,” MILCOM Tutorial, Baltimore, October 29, 2018.
3. Suresh Kothari, “How Mathematics can save us from catastrophic cyber attacks,” ICISS Tutorial, Bombay I.I.T., India, December 17, 2017.
4. Suresh Kothari, Ben Holland, “Learn to analyze and verify large software for cybersecurity and safety,” MILCOM Tutorial, Baltimore, October 25, 2017.
5. Suraj Kothari, GIAN 2017 – Analytical Reasoning and Experiential Learning with Applications to Software Analysis, MNIT, Jaipur, India, October 9-15, 2017.
6. Suraj Kothari, Workshop on Learn to Understand, Analyze, and Verify Large Software, National Institute of Technology, Patna, India, March 20-22, 2017.
7. Suraj Kothari, Ben Holland, Discovering Information Leakage Using Visual Program Models, MILCOM 2016, Baltimore, November 2, 2016.
8. Suraj Kothari, Jeremias Sauceda, Mission-Critical Software Assurance Engineering Beyond Testing, Bug Finders, Metrics, Reliability Analysis, and Formal Verification, ISSRE 2016, Ottawa, Canada, October 25, 2016.

9. Suraj Kothari, Ben Holland, GIAN 2016 – Managing Complexity, Security, and Safety of Large Software, MNIT, Jaipur, India, September 12-16, 2016. ***(Sponsored by the Government of India).***
10. Suraj Kothari, Ben Holland, Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework, ASE 2016, Singapore, September 4, 2016.
11. Suraj Kothari, Ben Holland, Practical program analysis for discovering Android malware**,** The Premier International Conference for Military Communications (MILCOM 2015), Tampa, Florida, October 28, 2015. http://www.afcea.org/events/milcom/Tutorials/tutorials.pdf
12. Suraj Kothari, Ben Holland, Hard problems at the intersection of cybersecurity and software reliability, International Symposium on Software Reliability Engineering (ISSRE 2015), NIST, Gaithersburg, Maryland, November 3, 2015.
13. Suraj Kothari, Ben Holland, Computer-aided collaborative validation of large software, IEEE Automated Software Engineering Conference (ASE 2015), Lincoln, Nebraska, November 3, 2015. http://ase2015.unl.edu/#computer-aided-collaborative-validation-tutorial
14. Suraj Kothari, Program Analysis and Reasoning for Hard to Detect Software Vulnerabilities, International Conference on Information Systems Security (ICISS 2015), Kolkata, India, December 17, 2015. http://www.iciss.org.in/iciss/tutorialsp.html

9. References

[1] Turing, Alan, “On computable numbers, with an application to the Entscheidungsproblem,” *Proceedings of the London Mathematical Society*, Vol. 2, no. 42, pp. 230-265, 1936.

[2] “Continuum hypothesis”, http://en.wikipedia.org/wiki/Continuum_hypothesis. Accessed October 18, 2019.

[3] “Gödel's incompleteness theorems”, https://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems Accessed October 18, 2019.

[4] “Paul Cohen (mathematician),” http://en.wikipedia.org/wiki/Paul_Cohen_(mathematician). Accessed October 18, 2019.

[5] Wei, Tao, Jian Mao, Wei Zou, and Yu Chen. "A new algorithm for identifying loops in decompilation." In *International Static Analysis Symposium*, pp. 170-183. Springer, Berlin, Heidelberg, 2007.

[6] Ahmed Tamrawi and Suresh Kothari, “Projected Control Graph for Computing Relevant Program Behaviors”, *Science of Computer Programming* Vol. 163, Pages 93-114, October 2018.

10. Appendix

10.1 Atlas Platform

RULER is implemented as a set of Eclipse plug-ins, which use Atlas as the reference implementation of XCSG.

Atlas uses Soot to produce the Jimple intermediate representation of Java bytecode. Jimple is a typed, three-address code, simplified representation of bytecode containing only 15 different instructions. More information about Soot and Jimple can be found at the Soot Sable homepage.

Atlas analyzes Jimple to import it into an eXtensible Common Software Graph (XCSG) compliant program graph. The following discussion is a very brief overview intended to help users get started, and focuses on XCSG as it pertains to Jimple.

In a nutshell, XCSG represents information from the AST, and conservative approximations of control flow and data flow. Most nodes are organized according to lexical nesting using Contains edges, the subgraph of which forms a tree. When visualized, Contains edges are displayed as nodes nested within nodes.

A typical path along Contains edges down to a Method goes through nodes with the following XCSG tags: Project -> Package -> Type -> Method. Fields are siblings of Methods, under the Type.

Possible invocations are represented between Methods by Call edges.

ControlFlow_Nodes and DataFlow_Nodes are embedded under Methods. All DataFlow_Nodes are immediately under a ControlFlow_Node (by a Contains edge). In XCSG for Jimple, ControlFlow_Nodes may appear under a TrapRegion.

ControlFlow_Nodes are connected via ControlFlow_Edges. DataFlow_Nodes are likewise connected by DataFlow_Edges. Operators are sub-kinds of DataFlow_Nodes. As translated by Atlas, each Field is represented by a single node, which results in an object-insensitive approximation. Other approximations may be added by RULER.

For additional help interpreting Smart Views, open the “Element Detail View” (Use *Window -> Show View -> Other*…). The Element Detail View displays the tags and attributes of any nodes and edges, which have been selected, whether they are in the text editor or in a graph. Buttons in the Element Detail View toolbar can be used to view a graph of the XCSG tag hierarchy relevant to the current selection.

More information about XCSG can be found on the XCSG wiki pages.

More information about Atlas APIs can be found on the Atlas wiki.

11. List of Symbols, Abbreviations, and Acronyms

- AC - Algorithmic Complexity
- API - Application Programming Interface
- CFG - Control Flow Graph
- CFR - Class File Reader
- ExCFG - Control Flow Graph with Exceptional Control Flows
- IO - Input and Output
- IPCG - Interprocedural Projected Control Graph
- JDK - Java Development Kit
- LCG - Loop Call Graph
- LTC - Loop Termination Conditions
- LTD - Loop Termination Dependency Graph
- PCG - Projected Control Graph
- RULER - Resource Usage vuLnerability identifiER
- SC - Side Channel
- STRU - Space/Time Resource Usage
- TDA - Targeted Dynamic Analysis
- TDG - Termination Dependence Graph
- XCSG - eXtensible Common Software Graph