Volume 1 — Technical Proposal — DHSST-LRBAA14-02

Reducing False Positives Reported By Static Code Analysis Tools

CSD.07 – Software Assurance

Indiana University
US Owned Entity

Technical Contact
Dr. James H. Hill (U.S. Citizen)
Indiana Univ.-Purdue Univ. Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202-5132
hillj@cs.iupu.edu; (317) 274-8527

Administrative Contact
Ms. Jean Mercer
Lockfield 2232
980 Indiana Avenue
Indianapolis, IN 46202-2915
spon2@iupui.edu

Co-PI
Dr. Rajeev R. Raje (U.S. Citizen)
Indiana Univ.-Purdue Univ. Indianapolis
723 W. Michigan Street, SL 280
Indianapolis, IN 46202-5132
rraje@cs.iupui.edu

Subcontractor
Dr. Adam Porter (U.S. Citizen)
University of Maryland, College Park
4125 A.V. Williams
College Park, MD 20742
aporter@cs.umd.edu

Period of Performance
August 1, 2014 – July 31, 2015 (base)
August 1, 2015 – July 31, 2016 (option 1)
August 1, 2015 – July 31, 2017 (option2)

DHS S&T Contact: Mr. Kevin Greene
DUNS Number: [PLEASE COMPLETE]

[Acknowledgement that Offerer register in Central Contractor registration]

(Compliance FAR Clause 52.222-54)

Authorizing Official--John W. Talbott
Asst. VP for Research Administration

Official Transmittal Letter

Table of Contents

EXECUTIVE SUMMARY

False positives are the dirty little secret of static code analysis (SCA) tools. Too often, SCA tool users have to sift through so many spurious error reports that they justifiably question the practical value of SCA tools. Our goal is to change this calculus by substantially reducing the number of false positives developers encounter, without dramatically affecting the number of true error reports. To do this we will develop and evaluate a highly automated learning system called Mangrove that incrementally and efficiently discovers, precisely characterizes, and publicly disseminates the program structures that lead today's state of the art SCA tools to falsely report errors in software. The system works by creating a repository containing highly simplified, abstracted and anonymized program snippets—derived from real programs—that cause current tools to falsely report errors. Next, the system uses machine learning techniques to mine these abstracted program snippets for rules that predict how likely it is that an individual SCA error report is a false positive. The program snippets and learned rules are then stored in a knowledge base that is periodically updated as additional information becomes available, allowing the knowledge base to improve its effectiveness over time.

This knowledge base will provide all major stakeholders in the SCA ecosystem—tool developers, tool evaluators, and tool users—with actionable insight into the strengths and weaknesses of SCA tools. Tool developers will better understand the fundamental software structures their tools can and can't handle correctly. Tool evaluators will be able to derive more complete and relevant test cases that compare multiple SCA tools against known hard to analyze code patterns. Tool users will be able to make informed tradeoffs between different tool configurations (i.e., some analyses find more errors, but only at the cost of greatly increasing false positives). Tool users will also use the knowledge base to triage tool reports, focusing initially on those reports with the lowest likelihood of being false positives and addressing remaining error reports only as time and resources permit. Such prioritization effectively reduces the high numbers of reported false positives users too often encounter today—allowing them to focus much more attention on resolving problems that are likely to be real. And finally, closing the loop, tool developers will benefit greatly, both from the availability of test cases derived from real software systems, and from increased usage data and user experience made possible only because users aren't handling mountains of false positive reports.

Building Mangrove requires broad experience in areas such as programming languages, software engineering, static source code analysis, machine learning, information processing, and software systems. To provide this experience, we have assembled a seasoned, broadly-skilled research team including Dr. James H. Hill and Dr. Rajeev Raje from Indiana University-Purdue University Indianapolis (IUPUI) and Dr. Adam Porter from the University of Maryland, College Park. Dr. Hill brings experience in domain-specific languages, model-driven architecture and static code analysis tool quality. Dr. Raje brings expertise in large-scale software systems and programming languages. Dr. Porter brings experience in applying machine learning algorithms to large-scale software system data to solve software engineering problems.

LANDSCAPE ASSESSMENT

Static code analysis (SCA) [Novak10] is the process of analyzing a program's source code, without executing it, to finds flaws. SCA tools are supposed to help developers quickly identify weaknesses and flaws that might jeopardize the security and integrity of a software program [German03]. Although SCA tools can clearly aid in identifying a program's potential weaknesses (e.g., errors that might compromise the program's security), they are also known to generate very large numbers of false positives [Chou05]. Simplifying greatly, this happens because tool designers have historically insisted on sound and complete analyses at the expense of other practical qualities, such as having succinct error reports that promote human usability. That is, designers have often acted, in effect, as if it were better to incorrectly identify correctly written code as having vulnerabilities, than to allow a single true vulnerability to go undetected.

For example, Listing 1 below shows correctly working source code taken from a file in the Juliet [Juliet12] test suite. When we applied one commercial SCA tool to this code it incorrectly reported that the while loop iterates forever. Of course, even the most novice computer programmer can rapidly determine that the loop in fact exits after 10 iterations.

```
...
static void good1() {
  int i = 0;

  while (1) {
    /* FIX: Add a break point for the loop if i = 10 */
    if (i == 10) { break; }
    printIntLine(i);
    i++;
  }
}
...
```

Listing 1. Code snippet from Juliet that produces false positive in a commercial tool.

In preliminary work PI Hill et al. [Hill14a] has been evaluating the quality of SCA tools, attempting to understand and predict their behavior. As part of that work Hill has evaluated several commercial SCA tools against the Juliet test suite. For this data set he has found that (1) a large majority of the SCA tool error reports were actually false positives; (2) the number of false positives was positively correlated with the number and type of potential flaws being searched for (*i.e.*, the more flaws the SCA tool was configured to check for, the more false positives it generated); and (3) the SCA tools often produce cascading flaws that resulted in numerous false positives with essentially the same root cause.

For example, Hill found that one commercial SCA tool disabled a certain analysis by default. He learned that the vendor did this because the analysis's implementation propagated information in such a way that any single issue would cause numerous other program sites to be incorrectly flagged as having errors. This behavior essentially made the analysis unusable in practice and therefore the vendor disabled it (personal communication, June 10, 2014).

As highlighted above, false positive reports are exceedingly common across all SCA tools. Most flawed error reports, however, are not so easy to analyze and dismiss as incorrect [Chou05]. Again, for example, in PI Hill's preliminary work, on multiple occasions he spent hours trying to determine why a commercial SCA tool identified a specific code file as having particular flaws, only to eventually determine that the report was a false positive [Hill14a]. In fact, due to the large number of false positives reported, developers often have to choose between spending large amounts of time searching for vulnerabilities that do not actually exist, or ignoring reported vulnerabilities because of the high likelihood that they will turn out to be false positives anyway. In all these cases, having an effective way to minimize or simplify programs could help developers manually analyze suspicious error reports.

From these examples and our experience, it's clear that if SCA tools are to have their strongest practical impact, then false positive reports must be substantially reduced. To do this researchers and engineers will have to (1) identify and catalogue a broad range of false positive reports; (2) understand the software structures that lead state of the art SCA tools to produce false positives; (3) develop test cases for measuring how well new and existing SCA tools perform against known problematic structures; and (4) and disseminate this information to SCA researchers, developers, evaluators, policy makers and end users to help evolve and improve the entire SCA tool ecosystem.

PROPOSED USE FOR DHS S&T

This project will develop a system called *Mangrove* aimed at reducing the number of false positives generated by SCA tools. Our aim is to provide stakeholders in the SCA ecosystem with information that can help them systematically, substantially, and effectively reduce the number of false positives they must deal with. We will complete this goal by incorporating the following major functionality into Mangrove:

1. Simplifying and abstracting real source code files that lead to false positive reports;
2. Representing and storing the abstracted code in a language-, technology-, and platform-independent manner that protects the intellectual property (IP) rights and privacy of the original code's owners;
3. Developing machine learning approaches for classifying and categorizing abstracted code and tool combinations to discover code features that tend to result in false positive reports, and for predicting whether specific error reports are likely to be false positives;
4. Developing a data-driven learning strategy and system for incrementally improving the tools and results created in function 3 across numerous individual observations; and
5. Developing a system that integrates the above steps into an actionable knowledge base that characterizes code structures responsible for false positives; allows new reports to be filtered to remove likely false positives; allows external entities to add observations to knowledge base; and provides real-life test cases for static code analysis tool developers and evaluators.

We envision that Mangrove will provide value to many different stakeholders within the DHS Science and Technology (S&T) directorate. For example, SCA researchers, developers, and evaluators can use Mangrove to understand how specific programming structures and tool configurations impact the number of false positives generated by a given SCA tool. They can also use Mangrove to filter and prioritize the reports generated by a SCA tool so that end users can focus only on highly probability flaws (i.e., deemphasize those reports that are likely to be false positives). Likewise, policy makers can use Mangrove data to assist in acquiring an SCA tool or tool combination, in defining coding standards, and more. Finally, organizations and end-users can use Mangrove to share information pertaining to false positives with other organizations. For example, the DHS can deploy Mangrove functionality into the SoftWare Assurance MarketPlace (SWAMP) (continuousassurance.org) to help it serve more effectively as the central knowledge base for false positive data collection, triage, and education. In fact, we believe that once fully developed and evaluated Mangrove will be most effective running in the context of the SWAMP. This is because the SWAMP is designed to process large numbers of programs, running them through large numbers of tools. These runs could provide an incomparably rich data set about the specific errors flagged and not flagged by different tools, and about the program structures that led to those reports.

TECHNICAL CONCEPT

Task 1. Extracting and Anonymizing False Positives

The first Task in our approach is to extract abstracted and anonymized code snippets from real source code that has led to a false positive report. These code snippets are intended to be minimal (*i.e.*, removing any statement from the snippet will not change the tool's false positive determination) so that humans can more easily reason about the cause of the false positive. In the initial stages of this effort, we will assume that we have an "oracle" that correctly determines whether a given error report for a given SCA tool applied to a given program file represents a true positive or a false positive. Later tasks will relax this assumption. Such information will initially come from PI Hill's current study on the quality and behavior of SCA tools [Hill14a], human analysis by the PIs, and, later in the effort, from community participation via a crowd-sourced online mechanism along the same lines as StackOverflow (www.stackoverflow.com).[1]

To complete this task, we will use an approach based on *Delta Debugging* [Zellar02]. Delta Debugging is a systematic approach that removes text from error-producing software—thereby minimizing it—until what remains still causes the same error, but any further reductions do not. This approach was initially proposed as a way to minimize test inputs that led to program failures to make human debugging easier. In our case, however, we will use Delta Debugging to systematically reduce a source code file producing a false positive down to one or more minimal

[1] Because protecting IP rights is a key concern we envision limiting access to programs and their analyses on a program-by-program basis.

source code snippets that still produce the original false positive. By design, the Delta Debugging process also allows us to recover additional code snippets, slightly different from the minimal set just described, that instead do not produce the false positive. These two sets of slightly different snippets, some of which result in the false positive report, some of which do not, will later be mined to learn the essential program features that led to the initial false positive report.

For example, going back to the program in Listing 1, it's easy to see that the call to the `printIntLine(i)` function has no bearing on the SCA tool's determination that the while loop is infinite. It can therefore be removed without changing the tool's determination. Delta Debugging automatically detects examples like this, and removes the superfluous (from the SCA tool's perspective) function call.

```
void m0() {
  int v0 = 0;

  while (1) {
    if (v0 == 10) { break; }
    v0++;
  }
}
```

Listing 2. Code snippet anonymized using Delta Debugging.

One result of applying Delta Debugging to the code file from which Listing 1 was taken, is a small code snippet such as the one illustrated in Listing 2. As shown in Listing 2, most of the file's contents have been removed. Within the original `good1()` function, the comments have been removed, the function name `good1()` has been changed to `m0()`, the `m0()` function is no longer declared static, the variable `i` is now renamed `v0`, and the call to `printIntLine()` has been removed. Most importantly, when analyzed by the SCA tool, this snippet, which now looks quite different from the original program, still generates the same false positive that the while loop is infinite.

Basic Delta Debugging minimizes its input on a character-by-character basis. Advanced versions are language-aware [Misherghi06]. Entire lexical tokens, rather than individual characters, are manipulated during each phase, which greatly speeds up the approach. Depending on which languages our examples are written in, we may need to develop our own language-specific tools. Our initial efforts, however, can go forward without these language-aware tools because they are not necessary for our initial feasibility experiments. Last, as shown above, since specific variable names and user-defined function names are immaterial to most SCA tools, the snippet will also be obfuscated (*i.e.*, all user-defined naming is replaced with randomly-generated names). This is primarily done as an initial step in anonymizing the minimized code snippets to better protect the identity and IP of the source code's owners.

Task 2. Independent Representation of False Positives

The second Task is to represent the minimized code snippet in a platform-, technology-, and language independent manner. This will allow us to (1) compare code structures causing false positives across different systems and programming languages; (2) build a suite of test cases (similar to Juliet) for regression testing new versions of an SCA tool, benchmarking false positive rates, and determining if the same false positive exists across multiple SCA tools; (3) generate test cases that target the same false positive causes, but in different programming languages; (4) represent the essential cause of the false positive, and (5) further and more strongly protect the identity and intellectual property of the original source code's owners[2].

Leveraging our previous research experience in behavior modeling [Paunov06, Hill07a, Hill07b], we will develop and use domain-specific languages (DSLs) [Schmidt06] to model the code structures found in abstracted code snippets. The DSL will represent common programming language constructs such as functions, variables, expressions, etc. It will also include control and data flow constructs found in abstract syntax trees (ASTs). As we capture large numbers of minimized code snippets, we will evolve the DSL accordingly.

We will validate our abstract representations by transforming them back into source code. If the generated source code produces the same SCA tool result as the original source code, then we will have increased confidence that our abstracted models are valid. The tools we use for validating abstract representations will be also be used for auto-generating false positive test cases as discussed above.

Finally, these abstracted models will be published in a repository to document and catalogue a wide range of simplified and abstracted source code examples known to produce false positives in one or more SCA tools. We envision that this resource will serve as a kind of benchmark or challenge such that in the future SCA tools will have to report how well they fare against these examples.

[2] We believe that Mangrove users will be much more likely to contribute to the overall Mangrove project if they are submitting abstract, anonymized code models, rather than actual source code files.

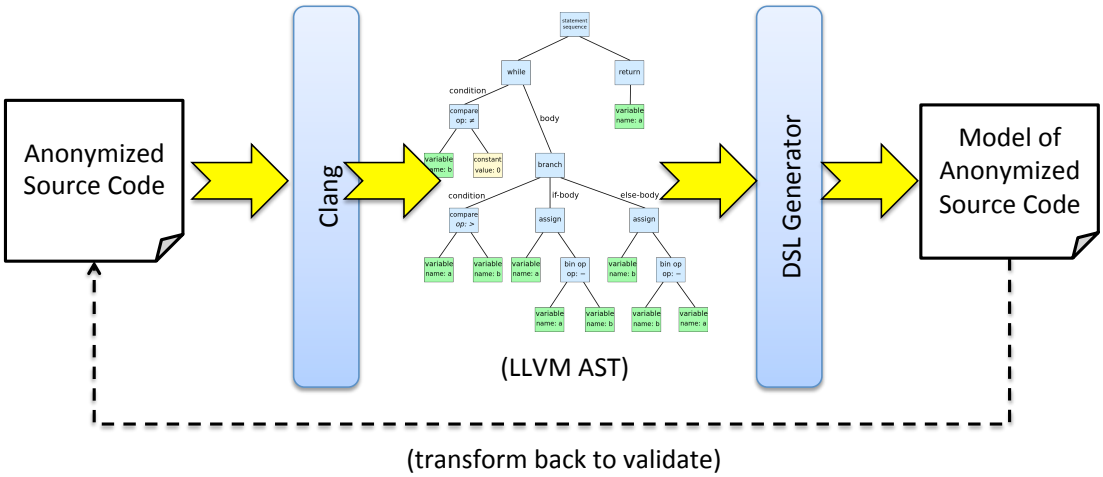

Figure 1. Workflow for generating abstract representations of false positives.

Figure 1 notionally illustrates how we will implement this task. To reduce technical risk associated with implementing a parser and generating the AST, we will first start with C/C++ source code and use LLVM [Lattner04] Independent Representation (IR) as the foundation for our DSL. We are selecting LLVM IR because (1) it is open-source and freely available; and (2) it supports the different constructs found in a programming language without being bound to a specific programing language, technology, or platform. As shown in the figure, we will start with C/C++ source code and use Clang [Lattner08] to transform the anonymized code into an LLVM AST. For other programming languages, such as Java and Python, we can use the appropriate LLVM frontend for that programming language, such as llvmpy (http://llvmpy.org) for Python.

We will then transform the LLVM AST into a model governed by our DSL and based on LLMV IR. Our DSL will extend the LLVM AST with additional information we need for cataloging and validating our models, such as originating programming language, source static code analysis tool, and software metrics related to the code snippet (e.g., cyclomatic complexity, source lines of code, and etc.). To validate our abstract representation, we will implement a simple compiler (or interpreter) to transform our DSL back into source code for the target programming language[3].

Task 3. Cataloging and Mining False Positives

The third Task will involve mining the abstract code models and associated error report information to learn rules, predicting the likelihood that a given error report represents a false positive. In principle, this learning activity can be based on a variety of measurable features, including the programming language constructs used in the underlying program and their data and control flow relationships, software metrics applied to the program, the SCA tool used, the

[3] All source code and any artifacts built from that source code will be segregated from public repositories unless the source code's owner expressly opts-in to its disclosure.

tool's configuration, and the type of error reported, other errors reported in the same source code file, and more. At this stage we do not know which of these data items and interrelationships will be the most useful for this prediction task and therefore we cannot limit ourselves, a priori, to a single machine learning technique or set of data.

However, in preliminary work [Hill14a], PI Hill has shown that different well-known software engineering metrics appear to be well correlated with the likelihood that SCA tools will report a false positive. Leveraging these preliminary results, we will initially focus on the software metrics he's identified (e.g., cyclomatic complexity, function points, max nesting, etc.), and we will use basic supervised machine learning algorithms for binary classification (i.e., a given error report is classified as either a true error or a false positive), such as classification trees [CART]. To evaluate this work we will compare the learning effectiveness achieved when using different software metric data on programs known to generate at least one false positive report. We will use this scenario as a baseline against which other approaches are compared.

As Tasks 1 and 2 progress, we will expand the set of measurable program features used in classification to include characteristics of the abstract syntax trees of the code snippets: graph metrics, such as vertex count, edge count, and the presence or absence of programming constructs, such as while loops, break statements, etc. We will also explore how different data features and different machine learning algorithms affect the costs and benefits of the overall approach. For example a large number of learning approaches may be viable for particular subsets of the available data, including Mining Association Rules [ASSOC], Latent Dirichlet Allocation [LDA], Neural Networks, and many others. Long term, we would like to identify the lowest cost, most accurate approaches, that also generate a false positive probability estimate (not simply the true error/false positive classification), whose results are the easiest for humans to interpret (interpretability will be more important to tool developers who want to understand why their tools generate false positives).

We believe that this general approach will greatly improve on PI Hill's current successes by expanding the set of machine learning algorithms used, the set of measurable data to which it's applied, and the number of observations underlying the analysis. In particular, the Delta Debugging approach described back in Task 1 will produce numerous variations of the original program. Considering only variations that compile without error, when multiple variations yield the same SCA tool outcome, then we will have effectively found multiple representations of the same false positive. When the new variations produce different SCA tool outcomes, then we have found multiple programs—slightly different from the first set—that do not produce the false positive. We expect these slight differences to quickly and powerfully highlight the key program structures leading to false positives.

We plan to store this information in a Git (git-scm.com) repository. This will provide us with robust mechanisms for storing, tracking, and distributing information and learned rules

describing the false positives. To draw an analogy, this information will be similar to the virus definition files that virus checkers use to represent currently known viruses. We will also create a web portal that integrates with our Git repo (similar to Github) where stakeholders (i.e., tool developer and tool users) can learn about known weaknesses of static code analysis tools. We envision something similar to MITRE's Common Weakness Enumerations (CWE) web portal (cwe.mitre.org).

As we begin to learn the features that best describe the program structures, tools, configurations, etc., that lead to false positives, we will also explore techniques for distinguishing the key information that uniquely encodes each particular false positive finding. For example, if Tool X always reports an error whenever the source code snippet contains a while loop with a break statement inside it, then there may be no need to record other features of the program or tool. These two pieces of information provide enough information to identify this particular false positive and its context. As we compile larger numbers of false positive examples, we will create N-tuple representations where key data measurements are used to represent different false positives in context. This N-tuple can then serve as a kind of DNA signature (or fingerprint) for the specific false positive in context. As we will discuss in Task 4, how this information will help us understand when additional information represents a new context, or represents another instance of a context we have already seen.

Task 4. Learning Over Time

In the earlier Tasks, we treat data collection and learning as a static activity performed on a single error report, generated by a single SCA tool, applied to a single code file. The real power of our approach, however, is that it could ultimately be applied in the context of the SWAMP. This means that we will have access to many of everything - many SCA tools, run in many configurations, applied to many programs and generating many error reports. Having such abundant data will enable us to draw conclusions and learn information in ways that are simply not possible when looking only at individual data points. To harness this potential however we must equip our system to learn over time. In particular, because error reports will arrive one by one over a long period of time, we must periodically regenerate our knowledge base to verify that our learning rules remain as accurate as possible. In addition, because in practice incoming reports will almost never be classified as true errors or false positives (Tasks 1 & 2 assume an oracle that provides this information), we must develop an alternate oracle strategy.

First, to better support learning over time, as new reports are added to Mangrove, we will automatically execute additional tests to expand the data set around this report. For example, given an error report, coming from SCA tool X, in configuration Y, applied to abstract snippet Z, we will re-execute tool X in multiple different sample configurations, and we will also execute other SCA tools in a sample of their configurations on abstract snippet Z. By systematically sampling these different scenarios, we can amass additional data that can isolate the cause of specific false positive reports to specific tools or tool configurations.

Next, we will create a system for providing information classifying error reports as true errors or false positives. Fundamentally, it's important to remember that this question is undecidable. If it were not, there basically wouldn't be any false positive problem to begin. Therefore, we will fall back to a manual approach that aims to identify specific error reports whose classification should provide the most new information to the knowledge base. Specifically, we will define similarity measures over abstract program snippets. Then when a new error report arrives we will transform the code associated with a new report into its abstract representation (as discussed above) and measure it's similarity to abstract snippets already in the repository. If very similar representations are found, then we assume that the current report is also likely to be a false positive (spotting checking manually every so often). If the current snippet is very dissimilar to those already stored, then we will analyze it manually—adding it to the repository as appropriate. The general idea is to quickly build a broad set of abstract snippets with manually verified error report classifications. Note: this approach does not rule out also using automated techniques and heuristics for identifying false positive error reports, but that will not be our initial focus.

Another potential idea to support this manual classification would be to create a Massive Online Open Course (MOOC) on SCA tools and to provide unclassified abstract snippets to students to be verified as homework assignments. Students taking the course would receive unclassified abstract snippets and be asked to verify whether specific error reports are true errors or false positives. Assuming, say 5, students review the same snippet we could essentially crowd source the classification determination. Such a MOOC could be beneficially be delivered as a tutorial to SWAMP users, teaching them how to use the SWAMP and how to interpret the data it generates. PI Porter has experience creating and delivering MOOCs, having recently delivered a MOOC on Android that had over 240,000 students enrolled (coursera.org/course/android).

Task 5. Mangrove as a Service

The fifth and final Task involves integrating the outcomes of the Tasks above into a single system named Mangrove that can be used for example (1) to triage false positives generated by a SCA tool—thereby reducing the number of flaw presented to the end-user for correction; (2) to query for code snippets similar to source code that has been flagged as having an error; (3) to produce reports detailing the strengths and weaknesses of given SCA tools and more. Once completed Mangrove will be capable of being deployed in both a private environment, such as a company, and a public environment, such as the SWAMP. Mangrove will allow its stakeholders (*e.g.*, tool developers, tool users, and educators) to extend it with new models of false positives, to validate existing false positives, and to learn which source code features control the false positive rate of static code analysis tools, all of which are unknown or poorly understood today.

Ideally, an end user will run an SCA tool on an existing code base and the SCA tool will generate a report about the code base (i.e. potentially containing false positive reports). The report is then passed through Mangrove to predict which flaws are likely to be real, which are false positives expressed with a level of certainty, and which cannot be classified. The real flaws

should be addressed by the end-user. The known false positives should be addressed by the end-user based on the level of certainty Mangrove gives the false positives. The unclassified reports (*i.e.*, the ones Mangrove has never seen before) are the ones the end-user can examine and used extend Mangrove's knowledge base on false positives.

The scenario above assumes Mangrove is being used in a single location by a single end-user (i.e., a single instance of Mangrove) and achieving even that would be a substantial improvement over the current situation. This, however, is only one use of Mangrove. Assuming the earlier Tasks go according to plan, we envision a federated version of Mangrove for creating distributed knowledge bases that pool even greater amounts of data. In this scenario, each instance of Mangrove will be able to communicate with other instances of Mangrove, and can share its knowledge base with other Mangrove instances as shown in the figure below. We envision the SWAMP to be the primary instance of Mangrove, similar to how MITRE is for primary source for information about Common Weaknesses Enumerations (CWEs).

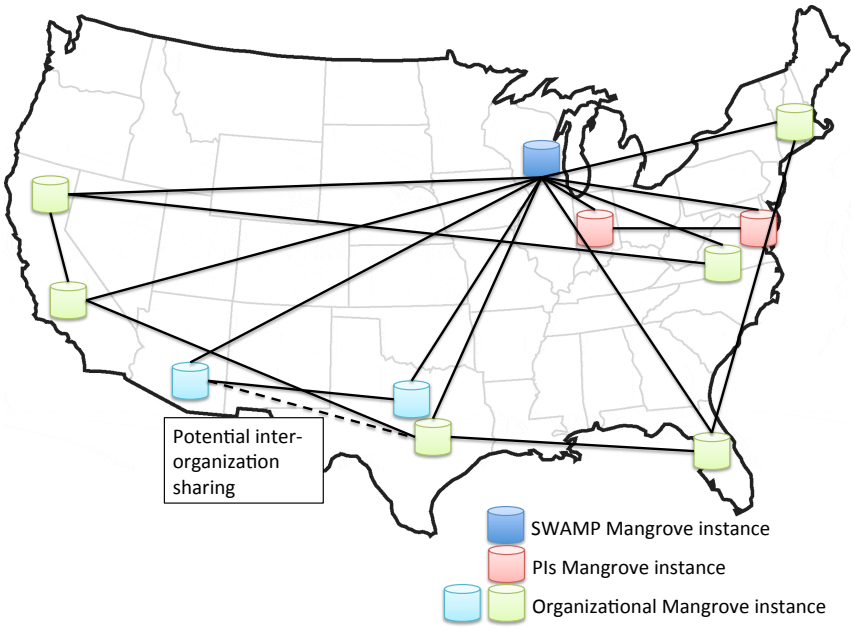

Figure. Mangrove as a Service—sharing between Mangrove instances.

As shown in this figure, different Mangrove instances will be able to communicate with other Mangrove instances. In some cases, Mangrove instances may be private, which means they are not sharing any information—a consumer that only updates its knowledge base with data from other Mangrove instances. In other cases, Mangrove instances may be organizational, which means there is sharing of information between Mangrove instances in the same organization. Last, Mangrove instances may be completely public—similar to the open-source philosophy. In cases when the Mangrove instance is private and organizational, we believe our approach to

minimizing and abstracting code snippets that cause false positives will encourage sharing with public Mangrove instances, such as the one located at the SWAMP.

To mitigate risk with implementing a solution from scratch, we will use Git as the underlying technology for the integration approach. We are selecting Git because (1) it is open-source and freely available; and (2) it supports self-contained distributed data stores, which would be false positive knowledge base in our case. This means its location will have a complete copy of the knowledge base, and not need to connect to a central location to use it. Based on our past experience [Hill08, Denton08] using Subversion (subversion.apache.org) to manage synchronization and coordination between software development tools, Git provides features out-of-the-box that will allow us to engineer a high quality solution, e.g., role-based access, controllable distribution, remote synchronization, and data management independent of all public and private knowledge bases.

References

[Juliet12] Tim Boland and Paul E. Black, “Juliet 1.1 C/C++ and Java test suite”, *IEEE Computer,* vol. 45, no. 10, pp. 88-90, Oct 2012.

[Zellar02] Zeller, Andreas, and Ralf Hildebrandt. "Simplifying and isolating failure-inducing input." *Software Engineering, IEEE Transactions on* 28, no. 2 (2002): 183-200.

[Misherghi06] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." In *Proceedings of the 28th international conference on Software engineering*, pp. 142-151. ACM, 2006.

[Denton08] Denton, T., Jones, E., Srinivasan, S., Owens, K., & Buskens, R. W. (2008). NAOMI–an experimental platform for multi–modeling. In *Model Driven Engineering Languages and Systems* (pp. 143-157). Springer Berlin Heidelberg.

[Schmidt06] Schmidt, Douglas C. "Guest editor's introduction: Model-driven engineering." *Computer* 39, no. 2 (2006): 0025-31.

[Hill07a] Hill, J. H., Tambe, S., & Gokhale, A. (2007, March). Model-driven engineering for development-time QoS validation of component-based software systems. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the* (pp. 307-316). IEEE.

[Hill07b] Hill, J. H., & Gokhale, A. (2007). Model-driven engineering for early QoS validation of component-based software systems. *Journal of Software*, *2*(3), 9-18.

[Hill08] Hill, J. H., White, J., Eade, S., Schmidt, D., & Denton, T. (2008, May). Towards a solution for synchronizing disparate models of ultra-large-scale systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems* (pp. 19-22). ACM.

[Hill14a] Lakshmi Manohar Rao Velicheti, Dennis C. Feiock, Manjula Peiris, Rajeev Raje, James H. Hill (2014, April). Towards Modeling the Behavior of Static Code Analysis Tools. 9th Cyber and Information Security Research Conference, Oak Ridge, TN.

[Lattner04] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (pp. 75-86). IEEE.

[Lattner08] Lattner, C. (2008, May). LLVM and Clang: Next generation compiler technology. In *The BSD Conference* (pp. 1-2).

[Novak10] J. Novak, A. Krajnc, and R. t'ontar, "Taxonomy of static code analysis tools," in *MIPRO, 2010 Proceedings of the 33rd International Convention*, 2010, pp. 418–422.

[German03] A. German, "Software static code analysis lessons learned," *Crosstalk*, vol. 16, no. 11, 2003

[Paunov06] Paunov, S., Hill, J., Schmidt, D., Baker, S. D., & Slaby, J. M. (2006, March). Domain-specific modeling languages for configuring and evaluating enterprise DRE system quality of service. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on* (pp. 10-pp). IEEE..

[Chou05] Andy Chou. 2005. False Positives Over Time: A Problem in Deploying Static Analysis Tools. Workshop on the Evaluation of Software Defect Detection Tools, Chicago, IL.

[CART] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. Classification and regression trees. Monterey, Calif., U.S.A.: Wadsworth, Inc.

[ASSOC] Agrawal, R.; Imieliński, T.; Swami, A. (1993). Mining association rules between sets of items in large databases. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data

[LDA] David M. Blei, Andrew Y. Ng, Michael I. Jordan, Latent Dirichlet Allocation, In Journal of Machine Learning Reserach 3(Jan):993-1022, 2003.