AFRL-RI-RS-TR-2016-059

SOFTWARE EPISTEMOLOGY

THE CHARLES STARK DRAPER LABORATORY, INC.

MARCH 2016

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 nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-059 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
JOSEPH CAROLI
Acting Technical Advisor, Computing and Communications 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)*
MAR 2016	FINAL TECHNICAL REPORT	DEC 2013 – SEP 2015

4. TITLE AND SUBTITLE

SOFTWARE EPISTEMOLOGY

5a. CONTRACT NUMBER FA8750-14-C-0056

5b. GRANT NUMBER N/A

5c. PROGRAM ELEMENT NUMBER 62788F

6. AUTHOR(S)

Jeffrey M. Opper

5d. PROJECT NUMBER T2ET

5e. TASK NUMBER SW

5f. WORK UNIT NUMBER EP

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

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

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

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

11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-059

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2016-0787
Date Cleared: 26 FEB 2016

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The effort developed a comprehensive approach for determining software epistemology which significantly advances the state of the art in automated vulnerability discovery. The approach applies an analytic sieve concept and a novel hashing scheme to a large corpus of open-source software to mine information that indicates the presence of pre- and post-fix conditions in program control flow, fully exploiting the hierarchy of abstraction and richness of data produced by the artifact extraction process, while taking advantage of the scalable computation capabilities present in TitanDB. The developed prototype software system is able to quickly analyze and compare software packages, demonstrating an ability to identify individual software components in a software system and track common vulnerabilities in software packages across large code corpora.

15. SUBJECT TERMS
Software epistemology, automated vulnerability discovery, large code corpora, analytic sieve, artifact generation, mining engine

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
U	U	U	UU	36	**STEVEN DRAGER**
					19b. TELEPHONE NUMBER (*Include area code*) **NA**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures iii

1. Introduction 1

1.1 Background 1

1.2 Overview 3

1.3 System Architecture 4

1.3.1 Harvester 5

1.3.2 Artifact Extractor 7

1.3.3 Object Ingestor 11

1.3.4 Relationship Integrator 12

1.3.5 Mining Engine 12

1.3.6 Analytic Sieve 13

1.3.7 Opcode Hash (OpHash) 14

2 Methods, Assumptions and Procedures 15

2.1 Implementation and Deployment 15

2.2 Theory of Operation 16

2.3 Corpus Creation 16

2.4 CVE Download 17

2.5 Vulnerability Detection 17

3 Results and Discussion 18

3.1 Demonstration One—Isolating a Vulnerability 18

3.1.1 Overview 18

3.1.2 Results 18

3.1.3 Summary 21

3.2 Demonstration Two—Full Package Vulnerability Assessment 22

3.2.1 Overview 22

3.2.2 Results 22

3.2.3 Summary 26

4 Conclusions 26

5 Recommendations 27

5.1 Open Questions 27

Bibliography 29

List of Symbols, Abbreviations and Acronyms 30

Table of Figures

Figure 1. SWE System Architecture 5

Figure 2. An example command line invocation of the Harvester. 6

Figure 3. Artifact Extractor JSON Object 8

Figure 4. Categories and types of artifacts 9

Figure 5. Artifact hierarchy 11

Figure 6. Ingestor command line invocation 11

Figure 7. The series of Gremlin [12] commands that invoke the Relationship Integrator. 12

Figure 8. Analytic Sieve 13

Figure 9. OpHash Map 14

Figure 10. Build inventory webpage 16

Figure 11. OpenSSL_1_0_1f (pre-fix) CFG 19

Figure 12. OpenSSL_1_0_1g (post-fix) CFG 19

Figure 13. Decompiled Dropcam binary (pre-fix) 20

Figure 14. Decompiled Dropcam binary (post-fix) 21

Figure 15. Lexumo OpenWebOS portal 23

Figure 16. Vulnerabilities and warnings in *elfutils* 24

Figure 17. Vulnerability details 25

1. Introduction

1.1 Background

The Charles Stark Draper Laboratory, a not-for-profit commercial laboratory, developed a comprehensive approach for determining software epistemology [1]. To this end, The Draper Laboratory developed a prototype software system to quickly analyze and compare software packages for similarity in composition. In this report, we discuss how our software epistemology system has demonstrated the ability to identify individual software components in a software system and to track common vulnerabilities in software packages across large code corpora. Draper's software epistemology system provides risk reduction to Air Force mission systems programs through detection and mitigation of vulnerabilities prior to deployment.

The Draper program's goal was to produce several proof-of-concept demonstrations within the planned 12 month term:

- Demo 1 - Demonstrate the ability to uniquely identify software based on a notion of canonical representation(s).
- Demo 2 - Demonstrate the ability to reverse engineer or uniquely identify AFRL prototype software from an in-house program.

Demo 1 objectives were demonstrated during our September 2014 review where we successfully detected the presence of the HeartBleed [1] vulnerability in Dropcam [2] firmware. We also demonstrated the efficacy of our *software analytic sieve* query pipeline for rapidly paring down query search spaces in large software corpuses. See Section 4 for details.

In Demo 2, unforeseen technical issues necessitated a change from the planned evaluation and testing with the AFRL-provided Real-Time Executive for Multi-processor Systems (RTEMS) [3]

---

[1] Epistemology is the study or a theory of the nature and grounds of knowledge especially with reference to its limits and validity

Approved for Public Release; Distribution Unlimited.

codebase. Instead, we demonstrated a successful analysis of the Open WebOS [4] operating system. Our analysis identified 24 known vulnerabilities (*i.e.*, vulnerabilities published in the MITRE Common Vulnerabilities and Exposures database [5]) in the latest release. See Section 5 for details.

During the execution of this effort, Dr. Suresh Jagannathan (a DARPA I2O Program Manager) was invited to attend a series of our monthly teleconferences; Dr. Jagannathan's research interests dovetailed with aspects of this program. Dr. Jagannathan successfully bootstrapped a DARPA program with intersecting goals, specifically, DARPA's Mining and Understanding Software Enclaves (MUSE). DARPA's MUSE [6] program builds upon the concept of software epistemology to investigate how large software corpuses can be analyzed to enable software repair and synthesis. Draper successfully proposed an epistemological and machine learning approach to the open MUSE BAA. Draper's proposed system, called DeepCode, extends the work performed here with advanced machine learning technologies. As proposed, DeepCode will apply machine learning over software corpuses at scale using deep neural networks, i.e., Deep Machine Learning, on high quality features computed from canonical representations of software, which would enable automated vulnerability detection, evolution and program repair.

Another indicator of the merit of this research is Draper's in-vitro decision to incubate a startup, Lexumo [7], which is developing a commercial Software as a Service (SaaS) vulnerability assessment platform based upon Draper's Software Epistemology (SWE) effort. Lexumo will, in turn, provide Draper with exclusive rights to use the Lexumo platform within the DoD and Intelligence Community (IC). Depending upon the specific customer requirements, Draper will either use the Lexumo platform as it exists (*e.g.*, unclassified vulnerability assessment of projects containing open-source software), or Draper will perform the necessary value-added engineering to extend the platform to accommodate custom features for DoD and IC customers.

In summary, the Software Epistemology project successfully demonstrated its core premise of identifying vulnerable code in modern complex software systems drawn from the wild by using

large code corpuses. During program execution, Draper invested significant in-kind internal research to perform risk reduction and technology exploration as well as incubated a commercial offering with Draper white-label support to DoD and IC customers—dramatically increasing the investment made by the Air Force. Finally, innovation continues on the DARPA MUSE program, where Draper's DeepCode effort is evaluating the application of Deep Learning on software features to support automated vulnerability identification and repair.

1.2 Overview

Draper's Software Epistemology approach originates from compiler intermediate representations (IR) of software. Because modern compilers all produce some form of IR during the compilation process, IR can be retrieved for any software package, and hence Draper's software epistemology system can utilize any and all open source code repositories to build a large, useful software corpus. Because many source packages reuse popular libraries, there is a high degree of commonality between the IR of different large software packages. For example, there is a small set of open source software libraries that are integrated into nearly all large software packages. As a result, given a new software package, Draper's software epistemology approach is highly likely to match a library or code fragment from that package to one already present in the epistemological database.

Previous efforts in software epistemology have focused on two contrary goals: first, small signatures that are able to identify malware that may have polymorphic presentation and multiple potential infection vectors, and second, large behavioral summaries for delta or regression analysis to ensure that software written against one version of a library can interoperate with another version of the same library. In the case of small signatures for malware, signatures must be highly compressible to allow for the distribution of a large number of signatures to a large number of vulnerable desktops. In the case of large behavioral signatures for libraries, the size of the behavioral signature may exceed that of the library itself, if the data is used to validate the correctness of a software system in development numbering in the millions of lines of code.

Draper's SWE effort has been to develop a scalable system that lies in a sweet spot between these two bodies of work. First, Draper's SWE effort looks at many large software projects. This allows for a high degree of parallelism in the search for similarities and differences between software packages. Consequently, we reduce the problem of determining software similarity to standard big data processing techniques such as map-reduce workflows and noSQL database queries. Second, Draper's SWE effort compresses large software projects into small sets of signatures, primarily representing their code reuse patterns, such that the signatures can be easily interpreted.

The ability to quickly and accurately identify software components—either from source code or machine binaries—enables the rapid identification of known software vulnerabilities, unsafe use cases, and hidden malware in complex embedded systems. In 2002, a NIST study [3] estimated the cost of faulty software to be between $22B and $60B in the US alone; with approximately half of the costs incurred from the labor and resources to mitigate the faults. SWE represents a revolutionary new approach to cyber security—in theory, by analyzing target software with the SWE platform, cyber security teams may be able to obtain a map of the software, with provenance to known examples of equivalent and similar software samples; associated metadata; and a list of all known vulnerabilities associated with the various software components without intensive human analysis.

1.3 System Architecture

The Software Epistemology prototype system adopts a workflow-based architecture where components of a toolchain are executed sequentially to build object code from downloaded software repositories; extract artifacts representing semantic relationships within the modules, functions, and basic blocks; store these artifacts in a distributed graph database; and rapidly pare down the search space to pinpoint vulnerabilities in systems of interest (Figure 1).

At a high level, there are three major subsystems that comprise the SWE prototype:

- Artifact Generation

- Mining Engine
- Analytic Sieve

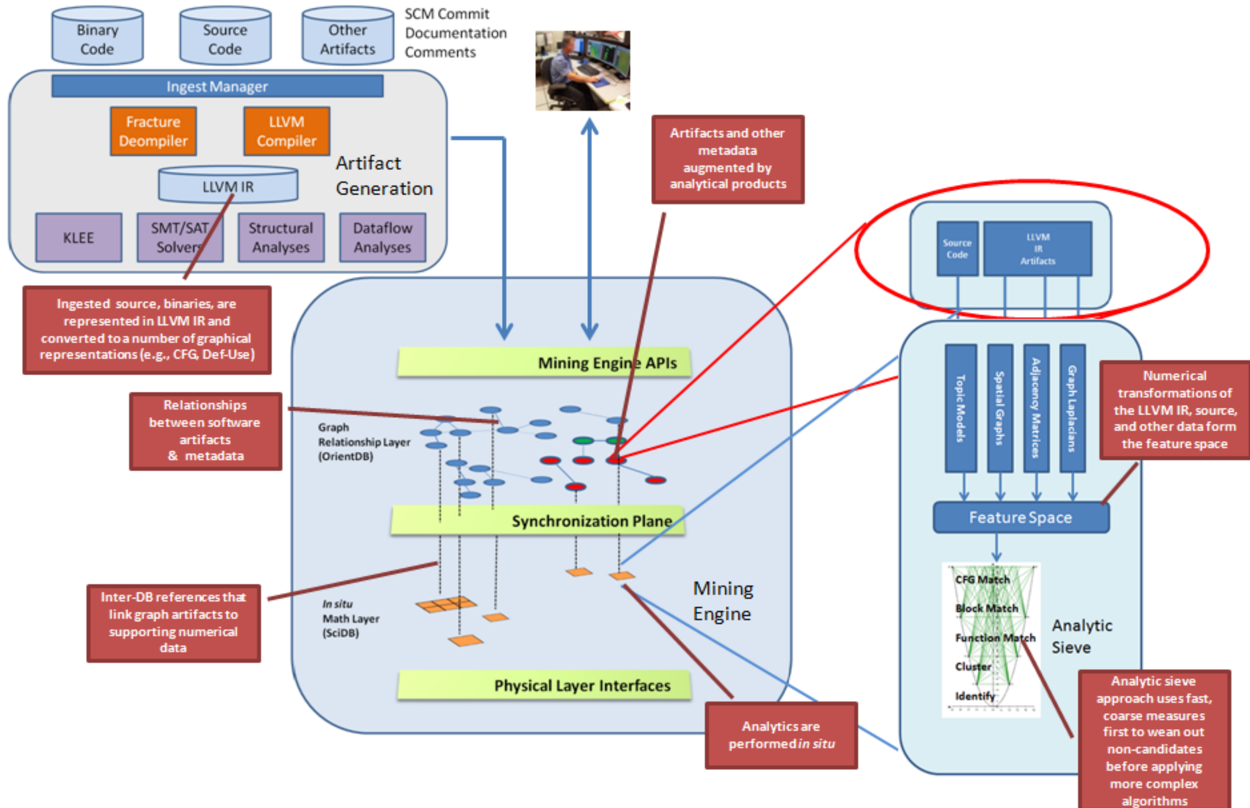

Figure 1. SWE System Architecture

Within the Artifact Generation subsystem reside the Harvester and Artifact Extractor tools. The Object Ingestor and Relationship Integrator tools reside at the boundary between the Artifact Generation and Mining Engine subsystems. The following subsections describe each component of the toolchain.

1.3.1 Harvester

The core requirement within SWE is that large open source packages are transformed into relatively smaller sets of artifacts that represent the call structure, control flow, and

opportunistically discovered semantic relationships between the modules, functions, and basic blocks within each project.

The Harvester's function is to build software revisions of software projects whose sources are stored within git repositories. For example, given a software project contained in repository *foo* with revisions $\{hash_1,\ldots,hash_k\}$, the harvester will produce k builds. The manifold build process is performed as a master/slave distributed process across nodes in a cluster.

The Harvester is a collection of python packages that build on the Yocto Autobuilder project, which in turn builds on the BuildBot project. Both of these projects develop Continuous Integration frameworks that automate software build processes.

The Harvester contributed to these frameworks by adding heuristics to attempt to identify the type of build required (*e.g.*, make, autoconf/automake, ant) and associate the appropriate *builder module* with the target. Further, in some cases, builds may generate transient products that are needed for the SWE ingest and artifact generation processes to succeed. To support this requirement, the linux's *strace* generalized debugger functionality is used in the builder modules such that an strace script identifies LLVM *clang* [8] system calls made during compilation. This information is used during a second build pass to allow the Harvester to capture files that would otherwise have been lost as temporaries in the build process.

A command line argument, see Figure 2, to the Harvester instructs it to either invoke the Artifact Extractor directly as the project is built or wait until the harvesting process is complete. In the second case, a separate command would be issued to start the artifact generation process.

```
python -mdcharvest.corpusTools.submitProject --config
/etc/puppet/modules/dcharvest/files/MasterConfig.cfg --builder
genericConfigure --submit --scrape <--limit X> --runDCAE --id <id tag> --
project http://plisl01.draper.com:8800/git/cntlm.git
```

Figure 2. An example command line invocation of the Harvester.

Approved for Public Release; Distribution Unlimited.

1.3.2 Artifact Extractor

The SWE Artifact Extractor takes LLVM IR compilation units (h.t.f., programs) as input and outputs JavaScript Object Notation (JSON) markup that encapsulates the artifacts associated with the program. In particular, the output of the Artifact Extractor consists of named objects, which are key value dictionaries that represent some item of interest related to the program being extracted, and typed edges that denote directional linkages between objects. All extracted objects and edges corresponding to a compilation unit (a LLVM module) live in a single nameless JSON [9] object. Each Artifact Extractor object is a named JSON object (see Figure 3) with a set of Attribute and they are typed by their "Type" attribute. When an object represents a piece of data or a variable, the type of the data or variable is represented by a "VarType" attribute.

```
{
  "8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58": {
    "DCAE": "1427481407",
    "Globals": [
      {
        "DefaultValue": "[12 x i8] c\"fib(%u)=%u\\0A\\00\"",
        "Name": ".str",
        "VarType": "[12 x i8]*"
      }
    ],
    "Name": "fib.ll",
    "Path": "tests\/fib.ll",
    "Type": "Module"
  },
  "8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58-fastfib": {
    "IsExternal": "false",
    "IsIntrinsic": "false",
    "Name": "fastfib",
    "Parameters": "i32",
    "Type": "Function"
  },
  "Edges": {
    "calls": [
      {
        "8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58-main":
"8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58-fib"
      },
    ],
    "dominates": [
      {
        "8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58-mainentry":
"8b69cd632024b6d8a4470331fa758b763a86b9775496561e5d2ee633d6f58-mainfor.cond"
      }
     ]
}
```

Figure 3. Artifact Extractor JSON Object

All edges corresponding to a module live in a single JSON object named “edges”. Each class of edges is a nested object named by the edge class composed of attributes of the form "in_name: out_name". This structure enables the representation of artifacts within a project as a connected graph that preserves the relationships between the objects.

Artifacts fall within one of the following categories Static, Dynamic, Derived, and Indirect as depicted in Figure 4. Static are those extracted from the LLVM IR without execution of the program under inspection. They describe program structure, inter and intra-module interfaces. For a given program, the SWE prototype extracts sets of functions, Call Graphs between those functions, traditional Control Flow Graphs (CFG) for each function, and dataflow graphs for each basic block within a Control Flow Graph (h.t.f. Use-Def). To simplify work on the analytics processor, pre-computation over graphs such as the Dominator Trees corresponding to CFGs are generated via standard LLVM library modules. Additionally, the SWE prototype mines program compilation artifacts for libraries, system calls, globally and externally available variables, constants, and known functions by walking the internal LLVM representations for a Program.

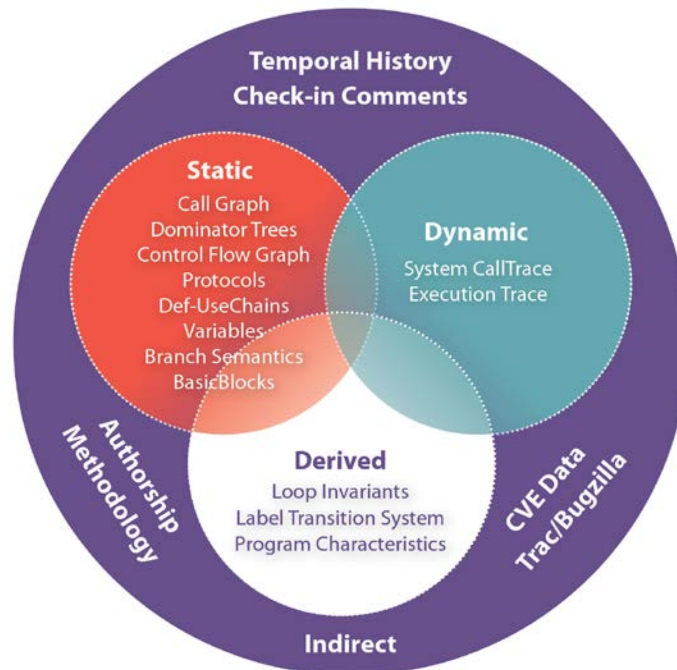

Figure 4. Categories and types of artifacts.

Table 1 describes the static artifacts that are generated by the Artifact Extractor at the time of this report.

Table 1. SWE static artifacts

Static Artifacts		
Name	**Description**	**Reason**
Call Graph (CG)	Directed graph of the functions called by a function.	Represents high-level program structure. Shows functions that are added, removed, or replaced.
Control Flow Graph (CFG)	Directed graph of the control flow between basic blocks inside of a function.	Represents function-level program structure. Shows basic blocks that are added, removed, or replaced.
Use-Def (UD) and Def-Use Chains (DU)	Directed acyclic graphs of the inputs (uses), outputs (definitions), and operations performed in a basic block of code.	Enables semantic analysis of basic blocks of code with regard to the input types accepted, the output types generated, and the operations performed inside a basic block of code.
Dominator Trees (DTs)	Matrix representing which nodes in a CFG dominate (are in the path of) other nodes. Comes in Pre (from entry forward) and Post (from exit backward) forms.	Highlights when the path changes to a particular node in a CFG. In compilers, DTs enable automatic parallelization analysis and other compiler optimizations.
Basic Blocks	The instructions and operands for inside each node of a control flow graph.	We can directly compare, and also produce similarity metrics between two basic blocks.
Variables	The types for any function parameters, local variables, or global variables. Includes a default value if one is available.	Provides initial state and basic constraints on the program. Shows changes in the type or initial value, which can affect program behavior.
Constants	The type and value of any constant.	See Variables.
Branch Semantics	The Boolean evaluations inside of if statements and loops.	Branches control the conditions under which their basic blocks are executed.

As described above, the static artifacts are graph-based and hierarchical in nature. This hierarchy is maintained in the ontological data representation of the Mining Engine. The artifact hierarchy is shown in Figure 5. The top of the artifact hierarchy is the Label Transition System (LTS). Each LTS node maps to a set or subset of functions and particular variable states. Under the LTS is the Call Graph (CG); each CG node maps to a particular function with a CFG. Each CFG node contains basic blocks, DTs, Use-Def (UD) / Def-Use (DU) chains, variables, constants, and other artifacts. Edges on the CFGs may contain loop invariants and branch semantics. Dynamic artifacts are mapped to multiple levels of the hierarchy, from an LTS node describing ranges of dynamic information down to individual IR instructions.

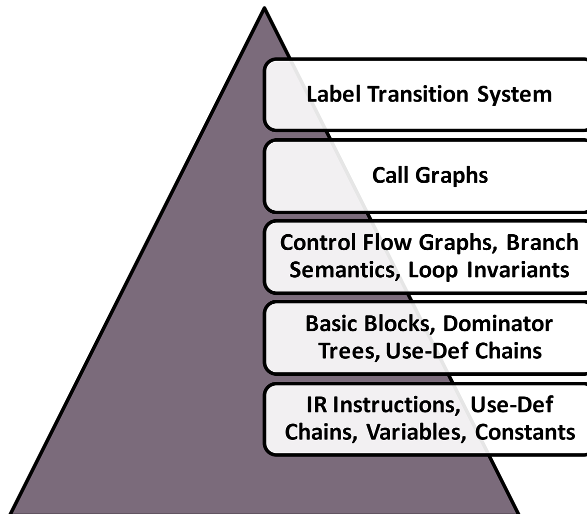

Figure 5. Artifact hierarchy

1.3.3 Object Ingestor

The SWE Object Ingestor imports, from a collection of JSON objects created by the Artifact Extractor, graphs representing the calls, control flow, and basic block instructions of an LLVM module into the graph database component of the Mining Engine. The ingest requires that a TitanDB keyspace has been created prior to invocation.

The ingest process is relatively straightforward. As the JSON objects are parsed, a connection is made to the database and queries are constructed to create vertices, edges, and their attributes in the named keyspace. Figure 6 illustrates the command line invocation of the Object Ingestor.

```
/usr/local/pyenv/versions/2.7.8/lib/python2.7/site-
packages/dcharvest/hdfs/ingestJSONCassandra.sh
"/user/corpus/<BuildID>/*/json/*.seq" /user/corpus/output/<BuildID>
<keyspace> <vertexTag>
```

Figure 6. Ingestor command line invocation

1.3.4 Relationship Integrator

The Relationship Integrator, invoked as shown in Figure 7, is a post-processing script that establishes relationships between each package and the modules, functions, and basic blocks that were present in each ingested tag for that package over its entire build history. These relationships are established through the creation of edges in the graph that represent the ownership hierarchy. This is extremely important as popular packages, such as OpenSSL [11], may have hundreds of tags representing the evolution of that software over a number of years. In each tag, files may be modified, introduced, and deprecated. The Relationship Integrator maintains this living history.

```
shell> /hdfs1/optnfs/titandb/bin/gremlin.sh
         \,,,/
         (o o)
-----oOOo-(_)-oOOo-----

// connect to DB
ks = <keyspace>
Conf = new BaseConfiguration();
Conf.setProperty("storage.backend", "cassandra");
Conf.setProperty("storage.hostname", "plisl01.draper.com");
Conf.setProperty("storage.cassandra.thrift.frame-size", "128");
Conf.setProperty("storage.cassandra.keyspace", ks);
g = TitanFactory.open(Conf);
// Load the dcri_titan_function.groovy script
load <local_dir>/deepcode-relationship-
integrator/dcri_titan_function.groovy
```

Figure 7. The series of Gremlin [12] commands that invoke the Relationship Integrator.

1.3.5 Mining Engine

The SWE artifacts are stored in an ontological graph layer using OrientDB [13] (initially) to preserve the semantic relationships between elements. Matrix representations of the graph-artifacts were also planned to be stored in a matrix-based math layer using SciDB [14] for efficient, distributed computation. The Mining Engine represents the conceptual unified query interface for the two database components in conjunction with an envisioned Synchronization Plane that kept relationships between data shared between the two instances intact (see Figure 1).

Initially, OrientDB performed nominally for small- to medium-sized data sets. However, as the size of the experiment datasets got sufficiently large, performance issues severely impacted ingest processing. By early 2015, it became clear that a different solution was required. After a brief evaluation of alternatives, the team replaced the OrientDB installation with a TitanDB [15]/Apache Cassandra [16] database and ported the toolchain to the new instance.

As of the writing of this report, all experimentation and demonstrations were performed using the Graph database component of the Mining Engine (*i.e.*, OrientDB or TitanDB).

1.3.6 Analytic Sieve

The Analytic Sieve is a more conceptual approach than a specific toolchain component, but there are framework components that support the approach. Early in our research, it became evident that a strategy needed to be adopted that maximized our ability to scale up to terabyte scale data sets.

The sieve concept takes the approach of beginning with fast, but effective, database queries that dramatically decrease the size of the search space. Building upon these initial queries, one then can apply increasingly more complex (and computationally expensive) queries to obtain the desired result (see Figure 8).

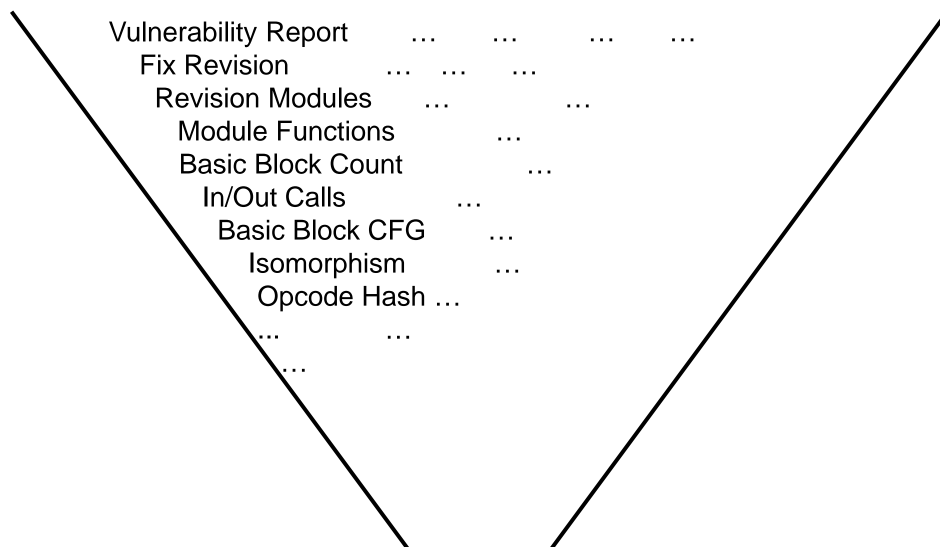

Figure 8. Analytic Sieve

In this depiction, one could start with the vulnerability report contained in the Common Vulnerabilities and Exposures (CVE) database to identify the versions of the software that displayed a particular vulnerability and the specific version that implemented the fix. Using the version number as a guide, one can immediately obtain the fix revision in the SWE artifact space to include the control flow graph that represents the region (pre- and post-fix). Using that data, the analyst can then examine any software system under test using hashing techniques and graph isomorphism to confirm or deny the presence of the vulnerability without relying on the version information alone. This is essential in cases where patches may have been introduced out-of-band and the version information in the source code does not match ground truth.

```
OpClassMap = {
  ('ret', 'br', 'switch', 'indirectbr', 'invoke','resume', 'unreachable') : 'Term',
  ('add', 'fadd', 'sub', 'fsub', 'mul', 'fmul', 'udiv', 'sdiv', 'fdiv',
   'urem', 'srem', 'frem') : 'Bin',
  ('shl', 'lshr', 'ashr', 'and', 'or', 'xor') : 'BitBin',
  ('extractelement', 'insertelement', 'shufflevector') : 'VectorOps',
  ('extractvalue', 'insertvalue') : 'Aggregate',
  ('alloca', 'load', 'store', 'fence', 'cmpxchg', 'atomicrmw',
   'getelementptr') : 'MemAddr',
  ('trunc', 'zext', 'sext', 'fptrunc', 'fpext', 'fptoui', 'fptosi', 'uitofp',
'sitofp', 'ptrtoint', 'inttoptr', 'bitcast', 'addrspacecast') : 'Conversion',
  ('icmp', 'fcmp', 'phi', 'select', 'call', 'va_arg', 'landingpad') : 'Other'}
OpClasses = ['Term', 'Bin', 'BitBin', 'Vector', 'Aggregate', 'MemAddr',
             'Conversion', 'Other']
Vals = [10 ** N for N in range(len(OpClasses))]
```

Figure 9. OpHash Map

1.3.7 Opcode Hash (OpHash)

A custom hashing scheme was developed in our SWE research to enable fast, but fuzzy, matching of basic blocks in a module or function. This scheme used a saturating histogram of LLVM IR opcode types encountered in a basic block. As an opcode is encountered in a basic block its type counter is incremented. Once the count reaches nine, it saturates even if other opcodes in the basic block map to that bin.

The LLVM Language Reference Guide groups opcodes into nine types:

- Other (O)

- Conversion (C)
- Memory Access and Addressing (M)
- Aggregate (A)
- Vector (V)
- Bitwise Binary
- Binary
- Terminator (T)

Figure 9 provides the mapping of opcodes to type as used in our OpHash scheme. The order of digits, as specified in the list, is OCMAVBBT. The resulting eight digit number is calculated for each basic block ingested into the system and maintained as an attribute of that object. Section 5 will specifically describe how the OpHash is used for the Demonstration One scenario.

2 Methods, Assumptions and Procedures

2.1 Implementation and Deployment

The SWE prototype consists of a number of open-source products and libraries combined with custom code. Table 2 provides a functional breakdown of the implemented system.

Table 2. Functional breakdown of SWE components

SWE Components		
Functional Area	**Item**	**Description**
Front-end	Buildbot	Meta-build framework for corpus ingest
	Modified *strace*	Preserves temporal build artifacts
	LLVM *clang* and plug-ins	Framework for Harvester, Artifact Extractor, and Relationship Integrator
Databases	PostgreSQL	Administrative database
	TitanDB/Cassandra	Primary graph store
	ElasticSearch	Supports Analytic Sieve
Analytics	Groovy, Gremlin, python	Scripting for analytic query processing
	SWE Analytic Sieve	Meta-query framework
Cluster Configuration and Maintenance	Puppet	Declarative language for system configuration and a cluster deployment tool

The SWE prototype was deployed on a Draper-owned 40-node compute cluster connected to the Cyber Enclave workstation area. Team members deployed software to the cluster using the Puppet tool referenced in Table 2. Several web-based tools were maintained to show cluster processing status. Figure 10 shows a snapshot of the build inventory on a specific day.

Figure 10. Build inventory webpage

2.2 Theory of Operation

2.3 Corpus Creation

The theory of operation of the SWE prototype is straightforward. First, internet-based repositories of open source software (*e.g.*, FreeBSD ports, GitHub, SourceForge, etc.) are mined

for projects (for the duration of this project we restricted our search to C and C++ projects for simplicity). These projects are mirrored locally (or staged) and the SWE toolchain is invoked (specifically the Harvester and the Artifact Generator) to build object code from which artifacts can be extracted and added to the TitanDB graph database. In addition to code artifacts, other metadata is extracted and used by the Relationship Integrator to build semantic links between graph nodes and includes build and revision histories, tags, and commit logs. As the ingest progresses, OpHashes are generated for each basic block consumed.

2.4 CVE Download

Separately, Common Vulnerabilities and Exposures data is downloaded from the web daily and loaded into PostgreSQL [17] and ElasticSearch [18]. Linkages of vulnerabilities to project, module, function, and basic block graph objects in TitanDB are maintained through scripts that are triggered during CVE processing.

2.5 Vulnerability Detection

Once a corpus has been established and the CVE data has been populated and linked, software of interest can be analyzed for known vulnerabilities. This analysis begins with a transformation of the source code to artifacts in a process that is identical to that performed when building a corpus. Then, provenance determination is performed to identify the known components (*i.e.*, open-source software libraries that are present in the corpus). Once components have been identified, initial vulnerability matches can be made using the links established during the CVE download process. These links are verified using OpHash matching of the basic blocks constituting the vulnerability segment in the corpus sample with the basic blocks contained in the same function in the software of interest.

Deriving the patch simply requires rolling forward to the fixed version (as specified by the CVE entry, if it exists) in the corpus and performing a *diff* between the vulnerable version of the impacted source file and the fixed version.

3 Results and Discussion

3.1 Demonstration One—Isolating a Vulnerability

3.1.1 Overview

By August 2014, the SWE toolchain was sufficiently mature for Draper to conduct the first real demonstration of program capabilities. For this demonstration, we focused on an analysis of the Heartbleed vulnerability in the OpenSSL open-source distribution. To isolate the vulnerability, in terms of its pre- and post-fix control flow graph changes, we ingested over 700 tagged builds of OpenSSL that spanned its full lifecycle to date. After isolating the fix delta, we attempted to perform the same process to determine if the firmware release present in an Internet-of-Things (IoT) streaming camera (Dropcam) was impacted by the vulnerability.

3.1.2 Results

Using the SWE analytic sieve approach, we first isolated the control flow graph artifacts from the pre- and post-fix versions. Figure 11 shows the CFG for the function dtls1_process_heartbeat in OpenSSL version 1.0.1f where the Heartbleed vulnerability was present. Figure 12 shows the CFG for the post-fix version 1.0.1g with the additional control flow for the bounds check present.

Approved for Public Release; Distribution Unlimited.

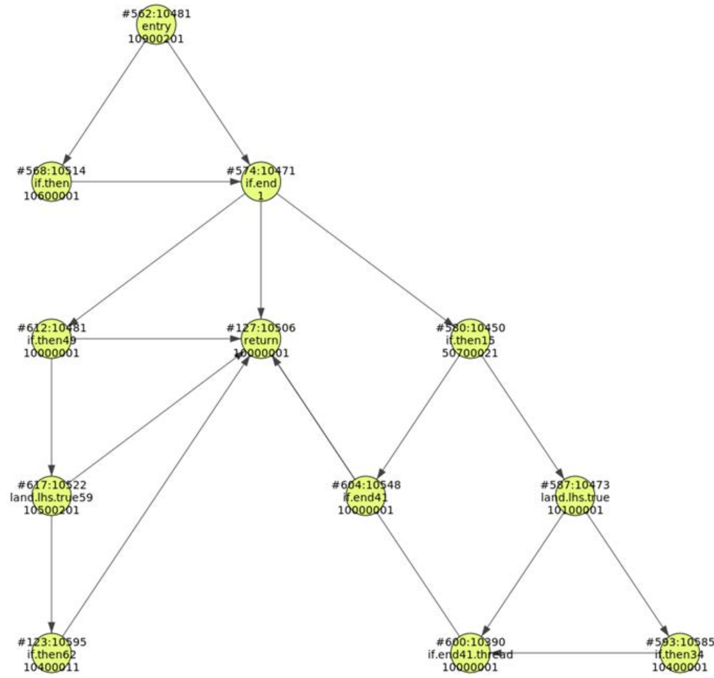

Figure 11. OpenSSL_1_0_1f (pre-fix) CFG

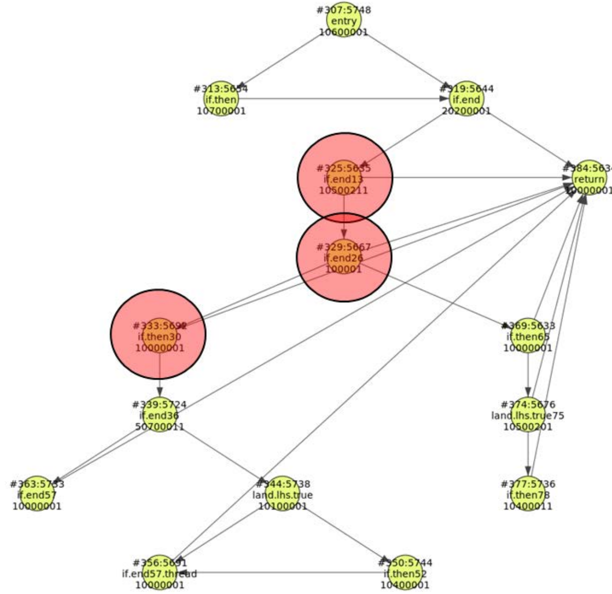

Figure 12. OpenSSL_1_0_1g (post-fix) CFG

In each CFG depicted, each node in the graph is annotated with the OpHash of the basic block contained in the node (the bottommost number). The green nodes in both graphs have matching hashes. The nodes highlighted in red introduce new OpHash values not seen in the pre-fix version.

We then decompiled different releases of the Dropcam firmware and perform the same solation process. In Figure 11 and 12, we see two CFGs with markedly different control flows (again detected by the OpHash). In Figure 12, we see evidence of control flow changes indicative of the Heartbleed fix (*i.e.*, #128, #131, and #143).

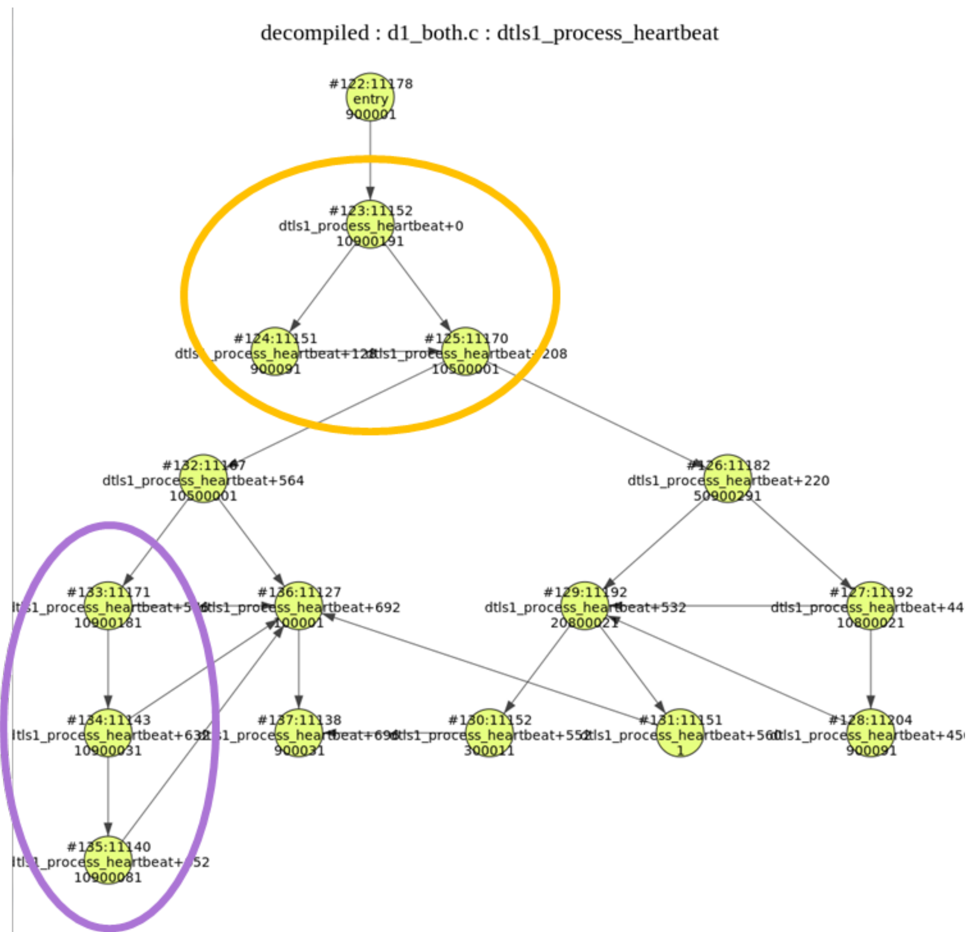

Figure 13. Decompiled Dropcam binary (pre-fix)

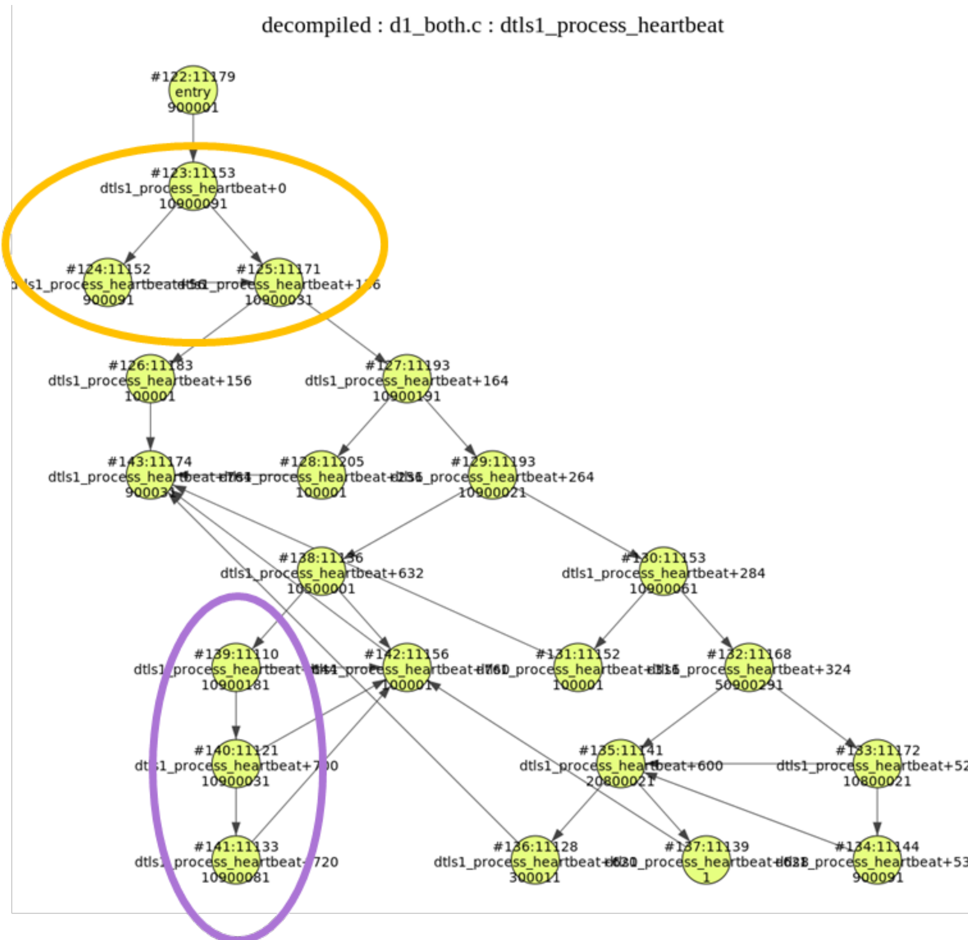

Figure 14. Decompiled Dropcam binary (post-fix)

One observation made in the comparison of the OpenSSL CFGs between Figures 13 and 14 is that the decompiled graphs have a larger number of nodes overall in each case. This is probably indicative of the IR being generated from code compiled originally at different optimization levels.

3.1.3 Summary

Demonstration One was successful on two levels. First, this was the first attempt to use the entire SWE toolchain on a single problem. Previously, portions of the toolchain were exercised in more of a unit testing mode. Finally, this demonstration validated the most basic SWE tenet—that a “big data” approach to software assurance can be bootstrapped from the ability to rapidly identify the essence of the delta between vulnerable code and patched code in a large corpus and then match the control flow in software under test.

3.2 Demonstration Two—Full Package Vulnerability Assessment

3.2.1 Overview

As mentioned in the Executive Summary, the SWE program resulted in two different transition stories. The first path, the DARPA MUSE program, extended the basic SWE approach from one of vulnerability identification to one of vulnerability identification augmented with repair and synthesis.

The second path was the decision to incubate a new company that would commercialize SWE technology for commercial interests with Draper retaining white-label rights for DoD and IC customers.

The second demonstration's goal was full package vulnerability assessment. Unforeseen technical issues necessitated a change from the planned evaluation and testing with the AFRL-provided Real-Time Executive for Multi-processor Systems codebase. The primary issue here was a custom build environment that would have required substantial changes to our front-end Buildbot infrastructure. Additionally, much of the code was self-referential and included no open-source components—rendering the known vulnerability search moot. For fiscal and practical reasons, we shifted strategies and attempted an analysis of the Open WebOS operating system for any known vulnerabilities. Open WebOS is an interesting target in its own right as it powers a number of IoT devices including HP TouchPads, LG Smart TVs, watches, and phones.

3.2.2 Results

While the analytics used in this demonstration are all SWE artifacts, we used the Lexumo customer portal for the Open WebOS analysis to take advantage of their rich user interface. Figure 15 shows the initial splash page for the customer Open WebOS project.

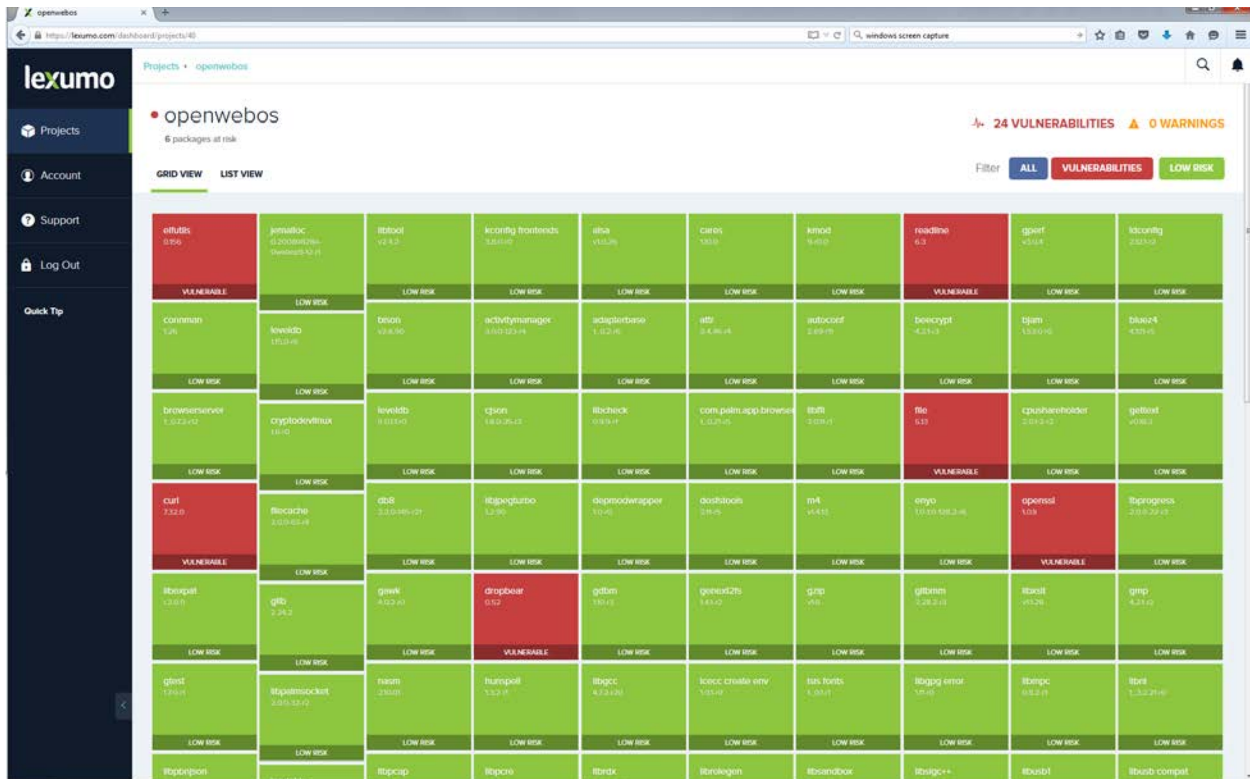

Figure 15. Lexumo OpenWebOS portal

This page provides all of the provenance details for the OpenWebOS package—detailing all included open-source libraries that were bundled in the package (including the version number). A vulnerability assessment of the current Open WebOS package using SWE analytics revealed that 24 known CVE vulnerabilities were present in the codebase. The affected libraries are shown in red in Figure 15 and are listed in Table 3.

Table 3. OpenWebOS vulnerabilities

Static Artifacts		
Library	**Version**	**CVE**
elfutils	0.156	CVE-2014-0172
readline	6.3	CVE-2014-2524
file	5.13	CVE2014-3478, CVE-2014-3480, CVE-2014-3587, CVE-2014-0207, CVE-2104-3479, CVE-2014-3487
curl	7.32.0	CVE-2015-3145
openssl	1.0.1i	CVE-2014-3571, CVE-2015-0286, CVE-2015-1792, CVE-2014-3567, CVE-2015-0209, CVE-2015-0205, CVE-2015-0204, CVE-2015-0206, CVE-2014-3572, CVE-2015-1789, CVE-2015-0287, CVE-2015-0288, CVE-2015-1791, CVE-1788
dropbear	0.52	CVE-2013-4421

Figure 16 shows the warnings and vulnerabilities present in the *elfutils* library. As Table 3, indicated, the CVE-2014-0172 vulnerability is present in addition to evidence that vulnerability CVE-2014-9447 existed in a prior versions of this library.

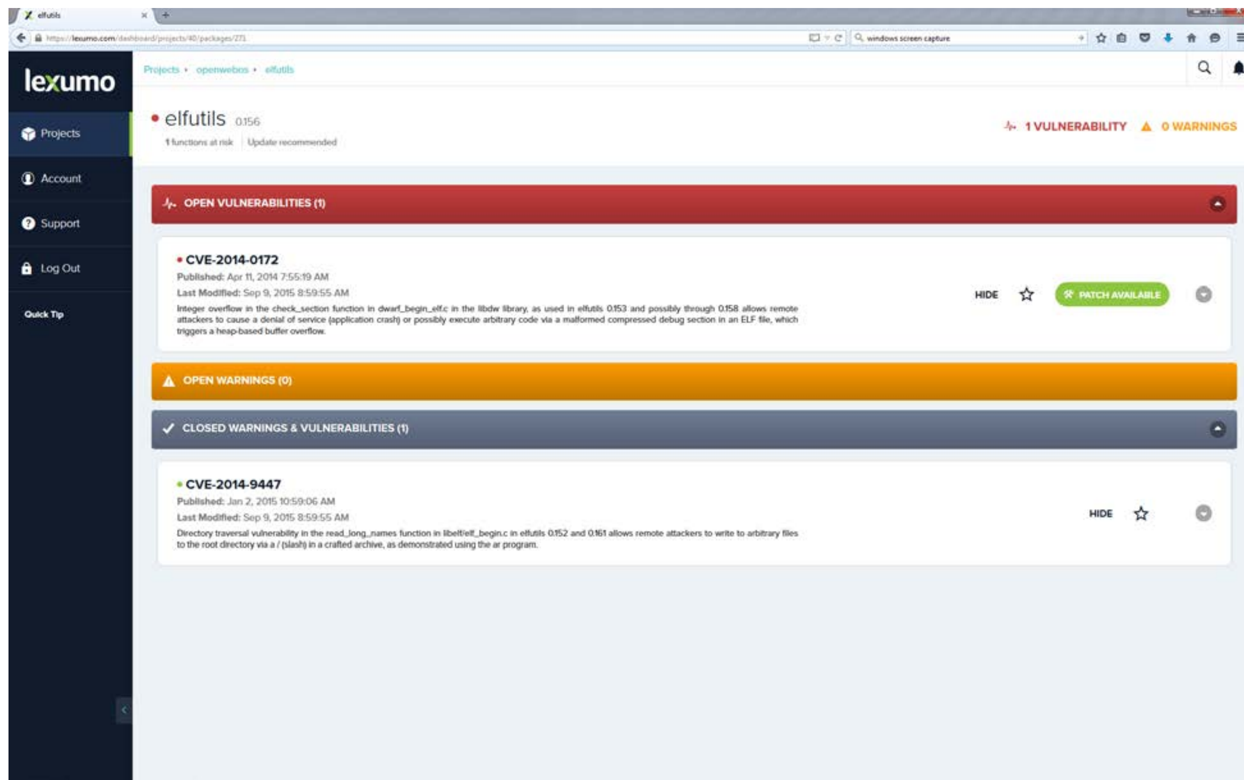

Figure 16. Vulnerabilities and warnings in *elfutils*

Figure 17 shows details regarding the vulnerability, evidence obtained during the analysis, and supporting information.

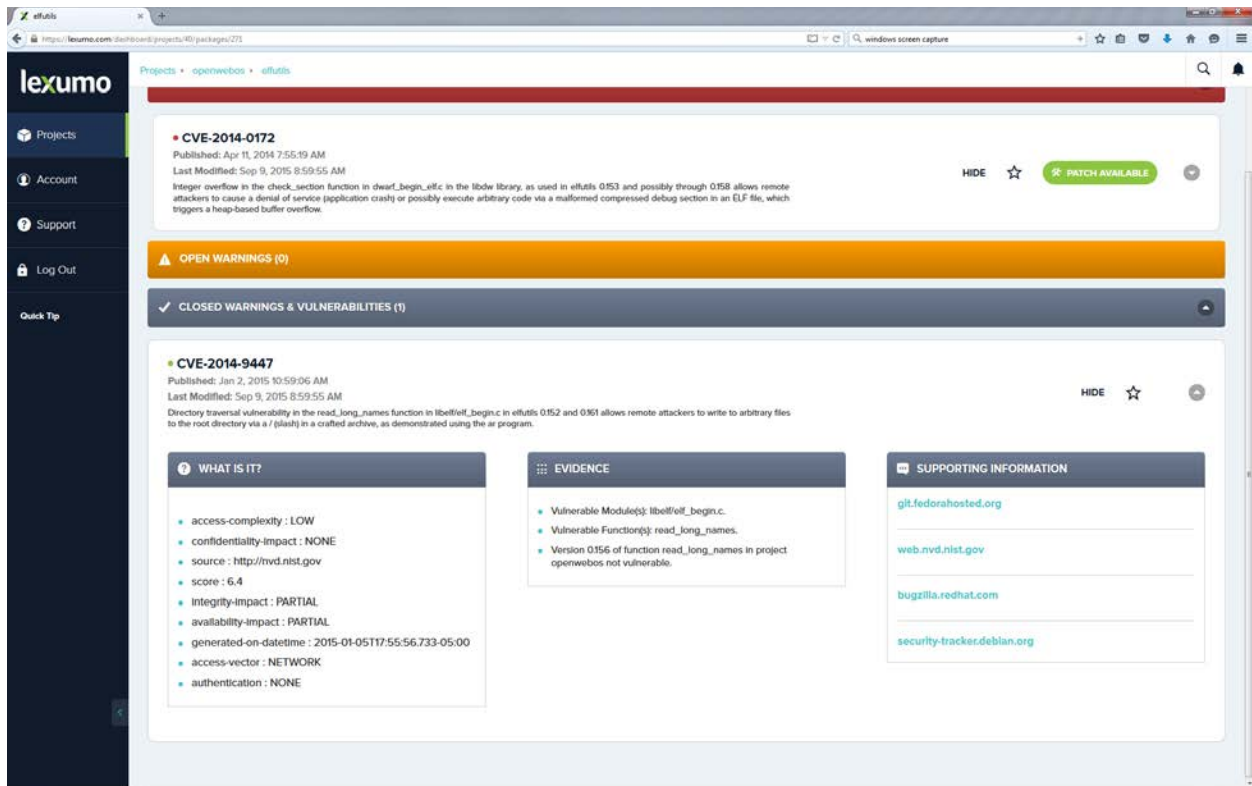

Figure 17. Vulnerability details

Finally, Figure 18 shows the flawed code segment and the scope of the available patch that fixes the flaw.

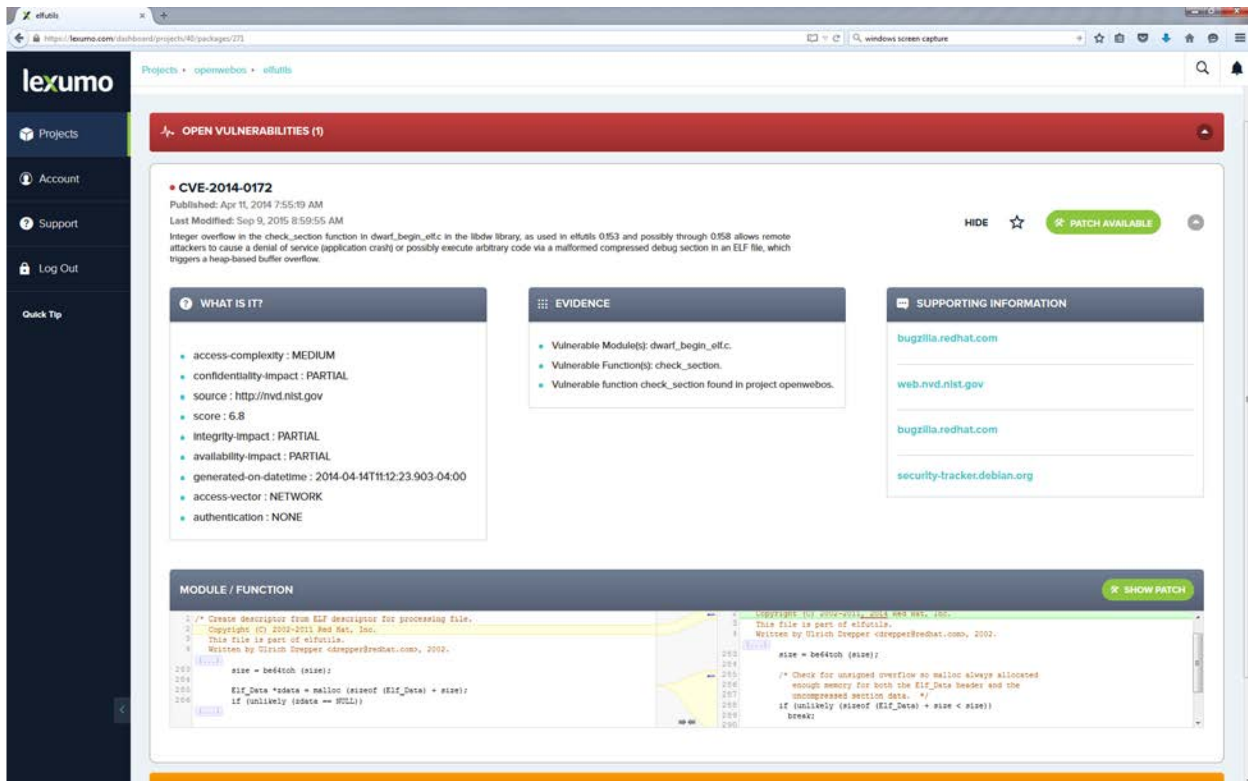

Figure 18. Vulnerability with patch

3.2.3 Summary

This demonstration, using the Lexumo portal for display purposes, successfully demonstrated the objective capability for this research—and analytic framework capable of full package vulnerability assessment.

4 Conclusions

Software Epistemology significantly advances the state of the art in automated vulnerability discovery—applying the analytic sieve concept and a novel hashing scheme to a large corpus of open-source software to mine information that indicates the presence of pre- and post-fix conditions in program control flow. The Draper Team's approach fully exploits the hierarchy of abstraction and richness of data produced by the artifact extraction process while taking advantage of the scalable computation capabilities present in TitanDB.

5 Recommendations

A measure of the success of this program is the two concrete transitions that have occurred over the period of performance. The first was the successful bootstrap of a related DARPA activity in the MUSE program. The second is the commercial entity that Draper is incubating that will enhance and extend SWE technology for commercial use. Both are significant wins for the Air Force. Lexumo will continue to develop and enhance the core SWE technology through venture funding. As the Lexumo platform matures, Draper has white-label rights to the platform to support DoD and IC customers. Draper's DeepCode effort on the DARPA MUSE program seeks to discover the fundamental nature of flaw patterns—applying Deep Learning algorithms to massive amounts of open-source software. This would remove the need for known vulnerability databases to guide the search and fundamentally change the way we approach software assurance.

5.1 Open Questions

There is a number of open research areas left to explore beyond the current effort. These include:

- Hash engineering
- Incorporation of other artifact types
- Additional static binary analysis
- Additional vulnerability database support

While the opcode hash is reasonably accurate and fast, it will saturate when exposed to large basic blocks. Other hashing schemes will need to be considered in these cases to maintain discriminatory capabilities.

SWE focused almost exclusively on the control flow graph as this artifact gave the best indicator of pre-and post-fix code structure. The LLVM compiler infrastructure provide a large number of other artifacts and Draper's DeepCode effort is starting to look at structures beyond the CFG for utility in software vulnerability assessment.

Draper has limited experience in the automated lifting of binary programs to a more structured language such as IR. However, static binary analysis is an open problem, and work of a more fundamental nature needs to be performed to generate Single Static Assignment (SSA) CFGs from binary. Our initial approach of inverting the LLVM code generator violates fundamental correctness invariants and was found to be a research dead end. Although Draper has explored more structured algorithms for static binary analysis, including approaches that show promise, we are not currently working on this problem.

Finally, additional vulnerability database coverage would provide more evidence of flaws that could be used in the assessment process.

Approved for Public Release; Distribution Unlimited.

Bibliography

[1] https://www.us-cert.gov/ncas/alerts/TA14-098A

[2] https://store.nest.com/product/camera/

[3] https://www.rtems.org/

[4] http://www.openwebosproject.org/

[5] http://cve.mitre.org/

[6] http://www.darpa.mil/program/mining-and-understanding-software-enclaves

[7] https://lexumo.com/

[8] http://clang.llvm.org/

[9] http://json.org/

[10] https://github.com/draperlaboratory/fracture

[11] https://www.openssl.org/

[12] https://github.com/tinkerpop/gremlin/wiki

[13] http://orientdb.com/orientdb/

[14] http://www.paradigm4.com/

[15] http://thinkaurelius.github.io/titan/

[16] http://cassandra.apache.org/

[17] http://www.postgresql.org/

[18] https://www.elastic.co/products/elasticsearch

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In Proc. *POPL*, pages 25—35, ACM, January 1989.

List of Symbols, Abbreviations and Acronyms

CFG	Control Flow Graph
CG	Call Graph
CVE	Common Vulnerabilities and Exposure
DT	Dominator Tree
IC	Intelligence Community
IoT	Internet of Things
IR	Intermediate Representation
JSON	Javascript Object Notation
LTS	Label Transition System
MUSE	Mining and Understanding Software Enclaves
RTEMS	Real-Time Executive for Multi-processor Systems
SaaS	Software as a Service
SSA	Static Single Assignment
SWE	Software Epistemology
UD/DU	Def-Use/Use-Def Chains (Dataflow Graph)

Approved for Public Release; Distribution Unlimited.