Impact Assessment for Vulnerabilities in Open-Source Software Libraries

Henrik Plate
SAP Labs France
Mougins, France
henrik.plate@sap.com

Serena Elisa Ponta
SAP Labs France
Mougins, France
serena.ponta@sap.com

Antonino Sabetta
SAP Labs France
Mougins, France
antonino.sabetta@sap.com

***Abstract*—Software applications integrate more and more open-source software (OSS) to benefit from code reuse. As a drawback, each vulnerability discovered in bundled OSS may potentially affect the application that includes it. Upon the disclosure of every new vulnerability, the application vendor has to assess whether such vulnerability is exploitable in the particular usage context of the applications, and needs to determine whether customers require an urgent patch containing a non-vulnerable version of the OSS. Unfortunately, current decision making relies mostly on natural-language vulnerability descriptions and expert knowledge, and is therefore difficult, time-consuming, and error-prone. This paper proposes a novel approach to support the impact assessment based on the analysis of code changes introduced by security fixes. We describe our approach using an illustrative example and perform a comparison with both proprietary and open-source state-of-the-art solutions. Finally we report on our experience with a sample application and two industrial development projects.**

I. Introduction

The adoption of open-source software (OSS) in the software industry has continued to grow over the past few years and many of today's commercial products are shipped with a number of OSS libraries. Vulnerabilities of any of these OSS libraries can have considerable consequences on the security of the commercial product that bundles them. The relevance of this problem has been acknowledged by OWASP which included "A9-Using Components with Known Vulnerabilities" among the Top-10 security vulnerabilities in 2013 [1]. The disclosure in 2014 of vulnerabilities such as Heartbleed[1] contributed even further to raise the awareness of the problem.

Despite the deceiving simplicity of the existing solutions (the most obvious being: update to a more recent, patched version), very often OSS libraries with known vulnerabilities are used for quite some time after a fixed version has been made available [2]. Updating to a more recent, non-vulnerable version of a library represents a straightforward solution *at development time*.

Unfortunately, the problem can be considerably more difficult to address when a vulnerable OSS library is part of a system that has already been *deployed* and made available to its users. In the case of large enterprise systems that serve business-critical functions, any change (including updates) comes with the risk of causing system downtime and new unforeseen issues.

[1]http://heartbleed.com/

The key question that vendors have to answer is whether or not a given vulnerability, that was found in an OSS library used in one of their products, could be exploitable, given the particular use that such product makes of that library [3]. If the answer is positive, an application patch must be produced and applied to all existing installations of the application. If the answer is negative, the update can be scheduled with the regular release cycle, saving the vendors the additional burden of developing an urgent patch and avoiding users the effort of applying it. The current practice of assessing the potential impact of a vulnerability in OSS is mostly a manual process that is time-consuming and error-prone. Vulnerabilities are typically documented with short, high-level textual descriptions expressed in natural language; at the same time, the assessment demands considerable expert knowledge about the application-specific use of the library at hand. The consequences of wrong assessments can be expensive: if the vendor incorrectly concludes that a given vulnerability is not exploitable, application users remain exposed to attacks. Conversely, if the vendor judges that the vulnerability is exploitable when in reality it is not, the cost due to developing, testing, shipping, and deploying the patch to the customers' systems is spent in vain.

This paper presents a pragmatic approach to support the decision making. Our approach consists in automatically producing (whenever possible) concrete evidences supporting the case for urgent patching. More in details *(i)* it automatically determines whether an application uses a library that is known to be vulnerable as it contains vulnerable code (assessment level-1), and *(ii)* it tries to determine whether the application actually makes use of the fragment(s) of the library where the vulnerable code is located (assessment level-2). We experimented our approach with a sample application and company internal development projects, and compared its results with state-of-the-art tools that have similar goals; in this paper we summarize our experience and the lessons learned.

The contributions of this papers are the following:

- Section II presents a novel approach that (1) supports the security assessment of applications that bundle open-source software with known vulnerabilities (assessment level-1) and (2) allows software maintainers to determine whether such vulnerabilities (if any) are actually relevant in a given application context (assessment level-2, based on the observation of concrete *executions*). The approach

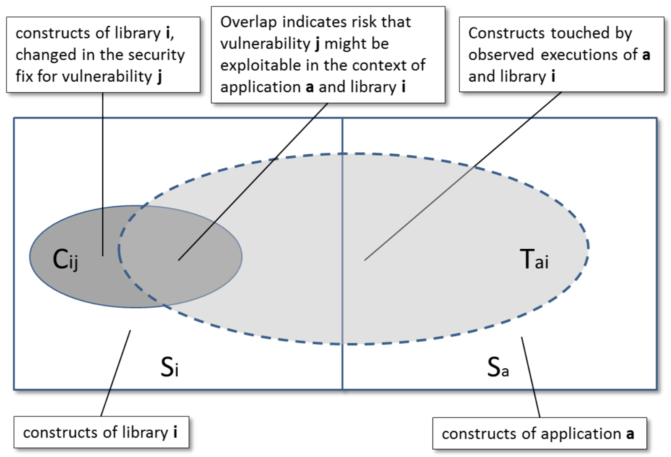

constructs of library **i**, changed in the security fix for vulnerability **j**
Overlap indicates risk that vulnerability **j** might be exploitable in the context of application **a** and library **i**
Constructs touched by observed executions of **a** and library **i**
Cij
Tai
Si
Sa
constructs of library **i**
constructs of application **a**

Fig. 1. Concept

is independent of programming languages and vulnerability types.

- Section III describes a prototypical implementation of our approach for Java, and demonstrates its use on a sample vulnerability (CVE-2014-0050). The prototype seamlessly integrates in the usual development workflow without requiring additional effort from developers.
- Section IV documents our experience using the prototype on a test application and two real-world applications. It also compares the results of our prototype with those of three state-of-the-art tools[2].
- Section V summarizes the lessons we learned regarding the remaining stumbling blocks that hinder automation of vulnerability assessment of systems that integrate open-source libraries.

II. Approach

A. Concept

In order to assess whether or not a given vulnerability in an OSS library is relevant for a particular application, we consider the *security patch* that is meant to fix the vulnerability. In practice, a patch is a set of changes that need to be performed in the source code of the library to remove the vulnerability. Our approach is then based on the following assumption:

(A1) Whenever an application that includes a library (known to be vulnerable) executes a fragment of the library that would be updated in a security patch, there exists a significant risk that the vulnerability can be exploited.

The underlying idea is that if an application uses library constructs[3] that would be changed when applying a patch, that is a strong indicator that the application might be relying on vulnerable code.

Therefore,

- we compare all constructs of libraries bundled in an application with the changes that would be introduced by the security patches to detect if vulnerable code is included; and
- we collect execution traces of applications and bundled libraries in order to detect whether the vulnerable code is not only included but actually executed.

Figure 1 illustrates the approach graphically for one generic OSS library i. The change-list C_{ij} represents the set of all program constructs of OSS library i that were modified, added or deleted as part of the security patch for vulnerability j. Change-lists can be computed as soon as a security patch has been produced for a vulnerability. In the case of responsible disclosure, patches can be assumed to be available at the time vulnerabilities become public. The set S_i is the set of all constructs of the OSS library i bundled in the application a whereas S_a is the set of all constructs belonging to the application itself. The collection of this information can be done at development-time, before the release of the application. If the change-list C_{ij} is not empty, then we can conclude that the application includes a library i with code vulnerable to j. This conclusion represents the first level of assessment (**level-1**) that our approach offers.

The trace-list T_{ai} represents the set of all program constructs, either part of application a or its bundled library i, that were executed at least once during the runtime of the application. The collection of traces can be done at many different times, starting from unit tests until the live system[4]. The intersection $C_{ij} \cap T_{ai}$ comprises all those program constructs that are both subject to security patch j and have been executed in the context of application a because of its use of library i. Following assumption **(A1)**, a non-empty intersection $C_{ij} \cap T_{ai}$ indicates that a newly disclosed vulnerability is highly relevant, due to the concrete risk of exploitability. An empty intersection, on the other hand, may result from insufficient coverage, hence, it does not necessarily imply that a vulnerability is irrelevant for the application at hand. This conclusion represents the second level of assessment (**level-2**) where we establish (if possible) whether the vulnerable code is executed. The overlapping of T_{ai} and S_a represents the execution coverage. The larger the overlap, the better the coverage, and the greater the confidence that constructs belonging to C_{ij} cannot be executed.

It is important to notice that program constructs modified in a security patch can be observed both in the vulnerable and fixed release of a library; in other words, though the body of a method is changed, the method signature itself is present in both releases. However we recall that our approach is meant to be used for answering to the question described in the introduction: should a vendor release an urgent patch to its customers because of a *newly* discovered vulnerability in a library included in an application? In this scenario we can safely assume that if S_i contains a construct changed in a patch, it

[2]Thereby focussing only on assessment level-1, since assessment level-2, to the best of our knowledge, is not supported by any other existing tool.

[3]We use the language- agnostic term "construct" to refer to structural elements such as methods, constructors, functions and so on.

[4]Trace collection on life systems is only possible if it comes with negligable performance penalities and does not impact system functionality.

must contain the vulnerable version as the vulnerability and corresponding patch were not existing when the application was developed and the constructs of S_i collected.

In other cases, if the library constructs and the traces are more recent than a patch, then it is necessary to identify the version of the library bundled in the application and compare it with the versions affected by the vulnerability.

From the description of the approach presented so far, it is clear that it is not immune to reporting false-positives and false-negatives. False-positives occur if a vulnerability is not exploitable, even when the vulnerable code is included and the intersection $C_{ij} \cap T_{ai}$ is not empty. Indeed, exploitability may depend on several other conditions; for instance, there might be, internally to the application, countermeasures (such as, the use of sanitization techniques or the adoption of suitable configuration options) that make it impossible to exploit the vulnerability. False-negatives, on the other hand, occur if vulnerabilities are exploitable despite an empty intersection $C_{ij} \cap T_{ai}$, which can result from insufficient coverage, a problem shared with most techniques relying on dynamic execution (as opposed to static analysis).

The silver-lining is that the approach is entirely independent of programming languages or types of vulnerabilities and, once the patch associated with a vulnerability disclosure has been processed, it can check large code bases and provide immediate results when comparing previously collected traces with change lists of newly published security fixes. In our experience, developers are easily persuaded that they need to update a library when they are presented with a non-empty intersection $C_{ij} \cap T_{ai}$, which is the proof that constructs that are subject to a security patch have been indeed executed. In any case, the information collected is valuable to simplify further analysis, complementing the high-level vulnerability description expressed in natural language that is typically found in publicly available vulnerability databases, such as the National Vulnerability Database[5].

B. Architecture

Figure 2 illustrates a generic architecture that supports our approach. Components depicted in white belong to the proposed solution and require an implementation, while components depicted in grey represent the solution's environment. A specific implemention for Java and related tooling is described in Section III.

The Assessment Engine on top is responsible for storing and aggregating the three sets of Figure 1. It also presents assessment results concerning the relevance of vulnerabilities to the security expert of the respective application.

The Patch Analyzer is triggered upon the publication of a new vulnerability for an open-source library. It interacts with the respective Versioning Control System (VCS), identifies all programming constructs changed to fix the vulnerability, and uploads their signatures (e.g., fully qualified method names, including formal parameters) to the central Assessment Engine. This change-list is built by comparing the vulnerable and patched revision of all relevant source code files of the library. The corresponding commit revisions can be obtained from the vulnerability database, searched in the commit log, or specified manually.

[5]https://nvd.nist.gov/

The Runtime Tracer collects execution traces of programming constructs, and uploads them to the central engine. This is achieved by injecting instrumentation code into all program constructs of the application itself and all the bundled libraries. The instrumentation can be done dynamically, during the actual runtime, or statically, prior to the application's deployment. The former can guarantee the tracing of all programming constructs used at runtime including, e.g., libraries that are only linked at runtime or parts of the runtime environment. Its major drawback is the impact on application startup, in particular if many contructs need to be loaded before the applicaton becomes available to its users, as in the case of application containers. Static instrumentation does not impact the application startup time and can be used in cases where the runtime environment cannot be configured for dynamic instrumentation, e.g., in PaaS environments. However, it cannot guarantee the coverage of program constructs used at runtime.

The Application Source Code and Dependency Analyzer scan the code of the application and all its dependencies, identifies all their program constructs and uploads their signatures as well as an application identifier to the central engine.

Note that the above-described components run at different points in time. Source Code Analyzer and Runtime Tracer are expected to run continuously during different phases of the application development lifecycle and their results are kept even after the release of the application to its customers. Once the Patch Analyzer is triggered, typically after the release of a patch for a vulnerable library, the assessment result is immediately available thanks to the comparison of the newly collected change-lists with the previously collected traces.

III. Proof-of-Concept

This section describes our current implementation of the approach and the architecture presented in Section II. We illustrate its application to CVE-2014-0050 as case study.

A. Implementation

The current prototype supports the assessment of vulnerabilities of Java components and is implemented by using technologies that can be seamlessy integrated in typical Java development and build environments based on Apache Maven.

The Assessment Engine is realized by means of a SAP Hana database where the change-list, trace-list and program constructs are stored and manipulated. The results are accessible via a web frontend (see Figure 4).

The Patch Analyzer is implemented as a Java stand-alone application. It interacts with version control systems (VCS) (Git and Subversion in our current implementation) by means of the JGit[6] and SVNKit[7] Java libraries. Such libraries are

[6]http://eclipse.org/jgit/
[7]http://svnkit.com/

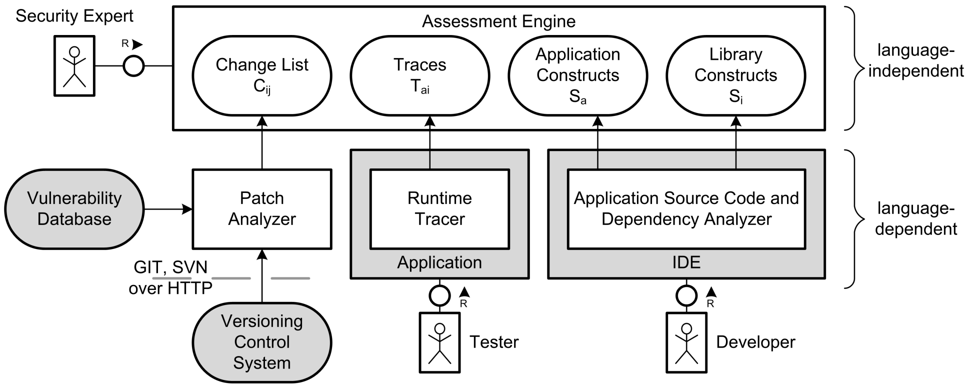


Fig. 2. Generic Solution Architecture

used to retrieve the code for the patch and vulnerable revisions (i.e., the revision preceeding the patched one). The ANTLR[8] library is used for building the parse tree of the Java files to be compared in order to obtain the change-list.

The Runtime Tracer requires the injection of code into each program construct of the application and its libraries. The instrumentation (both static and dynamic) is realized using Javassist[9]. Dynamic instrumentation was found suitable for unit tests, executed by individual developers and during continuous integration. Static instrumentation is instead suited for integration and end-user acceptance tests performed on dedicated systems. In particular, if the application is deployed in application containers such as Apache Tomcat, static instrumentations allows to avoid the performance impact on startup time, which is caused by the significant number of classes loaded before the container and its application become available. Moreover, static instrumentation is also useful in case Java Runtime Environment (JRE) options cannot be accessed or changed to enable dynamic instrumentation, e.g., when using Platform as a Service (PaaS) offerings. In either cases, whenever the application is executed, the instrumentation code added is responsible for collecting and uploading traces to the central engine. Note that the collection and upload is only done upon the first invocation of the respective programming construct which significantly reduces the performance overhead. The limited impact on application performance has as goal to enable the trace collection during everyday testing activities.

The Application Source Code and Dependency Analyzer are realized by means of a Maven plugin. It uses the ANTLR library to parse the Java source code of the application, and Javassist to analyze the byte code of all its dependencies. The result is the collection of the signatures of every program construct belonging to the application itself (S_a) or any of its dependencies S_i, all of which are uploaded to the central engine. The Maven identifier (composed of group id, artifact id, and version) is used as identifier of the application to set the context for the analysis, i.e., it represents the application a to define the sets S_a and T_{ai} of program constructs and trace-list (see Figure 1).

B. Case-study: CVE-2014-0050

The National Vulnerability Database (NVD) is a comprehensive, publicly available database for known vulnerabilities that are disclosed in a responsible manner, i.e., for which a patch has been made available at the time of the vulnerability publication. It also includes the information regarding affected products. In particular, the Common Vulnerabilities and Exposures (CVE) and Common Platform Enumeration (CPE) standards are used to identify the vulnerability and the affected products, respectively.

CVE-2014-0050[10] describes a vulnerability in Apache FileUpload [11] as follows: *"MultipartStream.java in Apache Commons FileUpload before 1.3.1, as used in Apache Tomcat, JBoss Web, and other products, allows remote attackers to cause a denial of service (infinite loop and CPU consumption) via a crafted Content-Type header that bypasses a loop's intended exit conditions."*

Upon disclosure of the vulnerability, any developer using Apache FileUpload needs to judge whether her application is affected. Current practices require her to rely on the textual description for taking a decision. However, it is not straighforward to assess whether the Java class MultipartStream (referenced to in the description) is used in the scope of an application. In fact it may be used either directly (i.e., instanciated in the source code of the application) or indirectly within other classes of the libraries which are directly used.

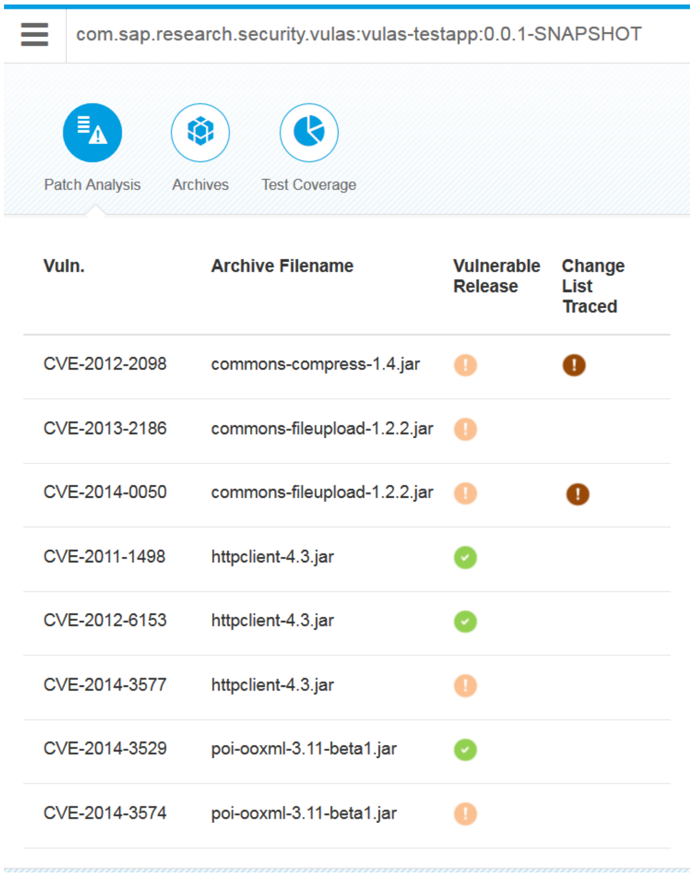
The application used in our case study is a sample Web application, com.research.vulas:vulas-testapp:0.0.1-SNAPSHOT, which performs various operations on compressed archives. Figure 3 shows assessment results for several vulnerabilities after the execution of both JUnit and integration tests of the Web application as well as the computation of change-lists. Column "Vulnerable Release" represents the result for assessment level-1, whereas column "Change List Traced" represents assessment level-2.

[8]http://www.antlr.org/

[9]http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

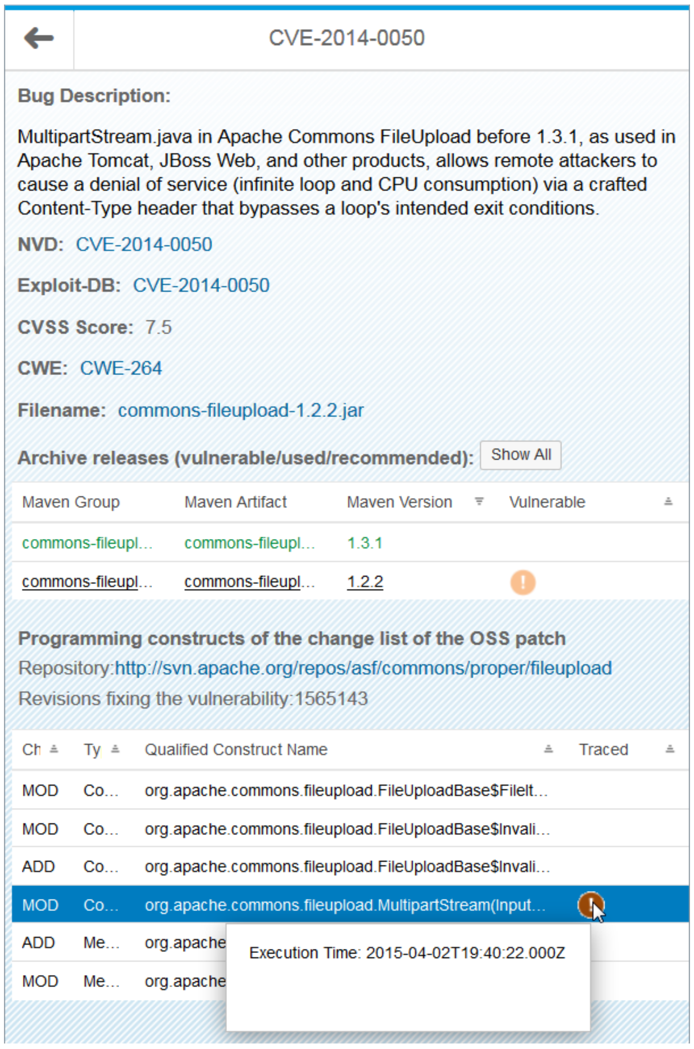
[10]http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0050

[11]http://commons.apache.org/proper/commons-fileupload/


com.sap.research.security.vulas:vulas-testapp:0.0.1-SNAPSHOT
Patch Analysis
Archives
Test Coverage

Vuln.	Archive Filename	Vulnerable Release	Change List Traced
CVE-2012-2098	commons-compress-1.4.jar	!	!
CVE-2013-2186	commons-fileupload-1.2.2.jar	!	
CVE-2014-0050	commons-fileupload-1.2.2.jar	!	!
CVE-2011-1498	httpclient-4.3.jar	✓	
CVE-2012-6153	httpclient-4.3.jar	✓	
CVE-2014-3577	httpclient-4.3.jar	!	
CVE-2014-3529	poi-ooxml-3.11-beta1.jar	✓	
CVE-2014-3574	poi-ooxml-3.11-beta1.jar	!	

Fig. 3. Analysis overview for the sample application



Bug Description:

MultipartStream.java in Apache Commons FileUpload before 1.3.1, as used in Apache Tomcat, JBoss Web, and other products, allows remote attackers to cause a denial of service (infinite loop and CPU consumption) via a crafted Content-Type header that bypasses a loop's intended exit conditions.

NVD: CVE-2014-0050

Exploit-DB: CVE-2014-0050

CVSS Score: 7.5

CWE: CWE-264

Filename: commons-fileupload-1.2.2.jar

Archive releases (vulnerable/used/recommended):

Maven Group	Maven Artifact	Maven Version	Vulnerable
commons-fileupl...	commons-fileupl...	1.3.1	
commons-fileupl...	commons-fileupl...	1.2.2	!

Programming constructs of the change list of the OSS patch

Repository:http://svn.apache.org/repos/asf/commons/proper/fileupload

Revisions fixing the vulnerability:1565143

Ch	Ty	Qualified Construct Name	Traced
MOD	Co...	org.apache.commons.fileupload.FileUploadBase$FileIt...	
MOD	Co...	org.apache.commons.fileupload.FileUploadBase$Invali...	
ADD	Co...	org.apache.commons.fileupload.FileUploadBase$Invali...	
MOD	Co...	org.apache.commons.fileupload.MultipartStream(Input...	!
ADD	Me...	org.apache	
MOD	Me...	org.apache	

Execution Time: 2015-04-02T19:40:22.000Z

Fig. 4. Analysis details for CVE-2014-0050

The first three vulnerabilities, including CVE-2014-0050, are marked as relevant because constructs part of the change-list (i.e., subject to the security patch) are included in the bundled library (assessment level-1). In the case of CVE-2012-2098 and CVE-2014-0050 constructs of the change-list have been executed; thus they are reported as relevant with assessment level-2 (as shown by the icon in the "Change List Traced" column). CVE-2011-1498, CVE-2012-6153, and CVE-2014-3529 are marked as not relevant since non-vulnerable releases of the respective libraries are used. For CVE-2014-3577 and CVE-2014-3574 the assessment result shows that the vulnerable release is used but no traces for the change-list were observed (assessment level-1).

More in detail, the table at the bottom of Figure 4 shows the change-list C_{ij} where i is Apache FileUpload and j the vulnerability CVE-2014-0050. In this case, the intersection $C_{ij} \cap T_{ai}$ is not empty, but contains the constructor of the Java class MultipartStream. The exclamation mark in the "Change List Traced" column highlights that its execution was observed –at the time shown in the tooltip– during application tests.

The Patch Analyzer computed the change-list using the URL of the VCS of Apache FileUpload and the revision number of the patch, which is provided by NVD in one of the references of CVE-2014-0050.

As the used program construct in Figure 4 was modified as part of the patch (marked as MOD in the figure), it is present both in the vulnerable and patched version of the library. Moreover, the experiment was done for a rather old vulnerability, hence, the traces are more recent than the security patch itself. As a result, it is necessary to identify the version of the library in use, and compare it with the ones affected by the vulnerability.

This is preferably done by means of Maven identifiers and, if this is not possible, CPEs. In particular, we search the Maven Central repository (i.e., the default repository for dependency management with Maven) for the SHA-1 of the archive from which a class was loaded at runtime and, if a match is found, we obtain the version of the used library as well as the information about all existing versions.

Versions affected by a vulnerability are identified by interacting with the VCS of the respective library. For that purpose, we analyze so-called tags, which are a common means for marking all repository elements that constitute a given relrease –at least in case of the widely-used versioning control systems Apache Subversion and CVS. In more detail, we identify all tags applied prior to the security patch in question, and parse the Maven project files that existed at that time.

In case of VCS other than Apache Subversion and CVS, we resort to the NVD for establishing the affected versions of a given vulnerability. The NVD uses CPE names, mainly composed of the vendor, product and version information, to identify all affected products. In order to establish if the used library is affected by the CVE we check if its version is among those listed for the affected CPEs. In general, the use of CPE names is considered as a fallback only, since the matching of CPE vendor and product names to Maven identifiers is ambiguous (cf. Section V).

Information about library releases is displayed in the upper table of Figure 4. It shows that the used library, i.e., Apache commons_fileupload 1.2.2, is indeed affected by the vulnerability: there existed a corresponding tag that has been applied prior to the commit of the security fix. It also shows the latest, non-vulnerable release, i.e., Apache commons_fileupload 1.3.1. By updating to the latest release, the risk of being vulnerable can be addressed.

As the Web application of our case study runs within the Apache Tomcat application container, we opted for the static instrumentation in order to avoid the impact of dynamic instrumentation on the container's initial startup time. The trace of the patched program construct was collected by using integration tests on the interface for uploading files of the instrumented application. Intuitively, we perceive that unit and integration tests are complementary means for collecting traces. In particular, the focus of unit tests on the business logic of fine-granular components does not cover components involved in the application's main I/O channels, many of which rely on OSS libraries, e.g., Apache FileUpload and HttpClient.

For CVE-2014-0050 an exploit exists as a Ruby script in the Exploit-DB, i.e., an archive of exploits for known vulnerabilities (http://www.exploit-db.com/exploits/31615/). By manually running it, we observed that assumption **(A1)** of Section II holds in our case study, i.e., the vulnerability is exploitable in the given application context even though only one program construct belonging to the change-list has been executed.

Other than assessing vulnerabilities, the prototype offers two other views (not shown here): The first shows all archives used by the application under analysis either because they have been declared using Maven or because they have been observed during application tests (i.e., classes where loaded from those archives). Archives whose SHA-1 is not known to the Maven Central are highlighted, and so are archives that have not been declared but whose execution was observed. The second view displays the function coverage of application constructs as described in Section II.

IV. Evaluation

We evaluated our approach against state-of-the-art solutions for identifying known vulnerabilities in software applications due to dependencies on OSS. In particular we considered OWASP Dependency Check (DC) and two commercial solutions.

Both commercial solutions are based on a central system, e.g., a web application, that based on application information is able to identify the relevant CVEs. On the contrary, OWASP DC runs locally (through a command line interface, a Maven plugin, an Ant task, or a Jenkins plugin) and downloads the CVEs information for the NVD. As a result, the first run of the tool may take several minutes on every system where it is used. Though they use different architectures, all the tools only check whether libraries with known vulnerabilities are included in an application by considering both direct and trasitive dependencies, i.e., all libraries an application project explicitly includes (direct) as well as those on which included libraries depend on recursively (transitive). Unlike our approach, neither OWASP DC nor the commercial tools considered assess whether the critical library-code is used.

More in details, OWASP DC checks whether a library with known vulnerabilities is included in an application project by inspecting direct and trasitive dependencies and collecting pieces of information (called evidences) about the dependencies that are then used to identify the CPE for the given library. The list of CVEs associated to the identified CPEs is then reported. To the best of our knowledge, also the two commercial tools we considered rely on a mapping from project dependencies to CPEs.

We perform the comparison at the example of an application we developed, and which can be used to compress archives with the bzip algorithm and offers such service as a web application (the same used in Section III-B). The reason for developing a test application was to include vulnerable libraries rather than the latest non-vulnerable ones in order to compare the findings of the tools. Note that we did not choose the libraries to include based on the CVEs they are affected by. As OWASP DC and the commercial tools only check whether a library affected by known vulnerabilities is included in the application project (without considering if the critical library-code is used), we limit the comparison to the first level of assessment we described in Section II.

The result of the comparison is shown in Table I where TP, FN, FP, TN stand for true positive, false negative, false positive, true negative, respectively.

In particular we observe

- one false negative for our prototype (CVE-2013-0248),
- one false negative and one false positive for OWASP DC (CVE-2013-2186 and CVE-2014-9527, respectively),
- four false negatives and two false positives for one of the commercial tools we considered (referred to as CT1 in the table and from now on), and
- one false negative for the other commercial tool (referred to as CT2).

The fact that CVE-2013-0248 is not reported by our prototype is due to the way our approach identifies vulnerable archives. As described in Sections II and III, this is done by checking whether a given archive contains a program construct (method or constructor) that is also part of a security patch for a given vulnerability. The security patch for CVE-2013-0248, however, only touches the default configuration

TABLE I
COMPARISON OF OUR APPROACH WITH OWASP DEPENDENCY CHECK

Vulnerability	Archive	Our Approach	OWASP DC	CT1	CT2
CVE-2014-0050	Apache FileUpload 1.2.2	TP	TP	FN	TP
CVE-2013-2186		TP	FN	FN	TP
CVE-2013-0248		FN	TP	FN	TP
CVE-2012-2098	Apache Compress 1.4	TP	TP	TP	TP
CVE-2014-3577	Apache HttpClient 4.3	TP	TP	FN	FN
CVE-2014-9527	Apache POI 3.11 Beta 1	TN	FP	FP	TN
CVE-2014-3574		TP	TP	TP	TP
CVE-2014-3529		TN	TN	FP	TN

file "using.xml" and the comment of the Java class "DiskFileItemFactory" (but not any executable code). As a result, our prototype does not identify the archive as containing vulnerable code, thereby resulting in a false negative.

The false negatives and false positives of the other tools originate from the fact that they identify vulnerable libraries by mapping archives (in particular their Maven identifiers) to the CPE identifiers (which are used in the NVD to list products affected by vulnerabilities).

The commercial tool CT1 reports false negatives for all CVEs related to Apache FileUpload. This hints at the fact that the tool failed at mapping the FileUpload library to its corresponding CPE.

OWASP DC is instead successfull at mapping the FileUpload library to its CPE as shown by the true positives for CVE-2014-0050 and CVE-2013-0248, however it fails at reporting CVE-2013-2186. Such CVE describes a vulnerability in the implementation of the DiskFileItem class in Apache FileUpload that allows remote attackers to write arbitrary content to any location on a server. This depends on the usage that the application server makes of the FileUpload library. As a result, the NVD assigned such vulnerabilities to several JBoss platforms without mentioning the origin of the vulnerability that lies in a method of the FileUpload library. Due to the fact that the CPE of Apache FileUpload is not mentioned as affected by CVE-2013-2186, OWASP DC fails at reporting it, thus resulting in a false negative.

CVE-2014-3577 is not reported by CT1 nor CT2. To the best of our understanding this is yet again to the failure of the mapping from the library to the CPE.

Until now we observed that relying on a mapping from libraries to CPEs introduces false negatives. However this may also introduce false positives as it is the case for CVE-2014-9527. The security patch for CVE-2014-9527 touches the Java class "org.apache.poi.hslf.HSLFSlideShow" (plus a test class and resource, which are not relevant). This class, however, is not part of the archive used in the application, "poi-3.11.beta1.jar". In fact, the Apache POI project comprises a number of different Java components (supporting different Office formats) among which the developer can chose. The vulnerable class is part of the component POI-HSLF, which is distributed as a separate JAR[12]. As the vulnerable Java class is not part of the archive, the vulnerability is not relevant in the scope of this application. However OWASP DC and CT1 map the used jar to the CPE `cpe:/a:apache:poi` that is not distinguishing among different parts of the POI project and listed in the NVD as product affected by CVE-2014-9527. In addition, we believe that CT1 does not correctly map the library release as it shows a false positive also for CVE-2014-3529 which is affecting releases older than the one in use.

We performed the comparison with OWASP DC also using two real world Java applications. The first application (referred to as App1 in the following) is a Maven project handling the communication functionalities of a bigger solution. The second application (App2) is a self-contained application realized as a Maven project composed of 11 modules, i.e., subprojects of the parent Maven project. App1 makes use of 22 distinct libraries while App2 uses 152 of them (if the same library is used across several modules we consider it once as it will be bundled once in the released application). The above numbers include libraries imported either as direct or transitive dependencies.

For App1, our prototype reported the use of non-vulnerable releases of libraries, in particular for the Catalina component of Tomcat and parts of the Spring framework, and no CVEs require further assessment. OWASP DC, on the contrary, reported a large number of CVEs, mainly due to the way it maps libraries to CPEs. In fact, when looking for evidencies the tool ends up in mapping the Catalina component to the entire Tomcat application in a vulnerable release, and the used parts of Spring to the entire framework. As a result, all CVEs affecting Tomcat and the entire Spring framework are reported. Though the vast majority can be easily assessed as non relevant, still it requires an additional effort compared to our approach.

For App2, our prototype showed that no vulnerable releases of the libraries are used. Instead OWASP DC reported two false positives for a library for processing of JSON data. In this case the tool failed at mapping the library to the correct CPE and thus included CVEs affecting several products making use of the TLS/SSL protocols which is not a functionality offered by the used library. OWASP DC also reports five CVEs affecting an SAP software. However these are again false positives as the mapping of a used library is done to the CPE of an unrelated SAP software.

After the detailled comparison of analysis results, we will now discuss the root causes of the false-positive and false-negative findings.

[12]http://poi.apache.org/slideshow/

The comparison was done for the assessment level-1, i.e., assessing whether an application includes a library that contains at least one construct that is also subject to a security patch. In other words we assess whether an application includes a library with vulnerable code (according to our assumption **(A1)**). The approach common to the other tools is to first identify a bundled library (e.g., its Maven identifier as in case of OWASP DC), which is then mapped to product identifiers (CPEs) affected by known vulnerabilities (CVEs). As a result, our approach is largely independent of CVE and CPE information provided by vulnerability databases, while the other tools fundamentally rely on the accuracy of CVE and CPE information. While we explain general problems related to information integration and information quality in Section V, the remainder of this section explains their consequences in terms of false-positive and false-negative findings when comparing the different tools.

With regard to false-positives reported by the OWASP DC and the commercial tools, it is possible to distinguish two cases. The first is related to the granularity of CPEs: Large, multi-module open-source software such as Apache POI are typically assigned to a single CPE, which ignores that application developers may chose only a non-vulnerable subset of the respective modules (cf. CVE-2014-9527 in Apache POI). The second is simply related to the mere problem of mapping human-provided identifiers to each other. This is the case of the false positives reported by OWASP DC for App2: the tool mapped the Maven identifier of the library for the processing of JSON data to the CPE of an Oracle product.

False-negatives can be reconducted to three main cases. As for the false positives discussed above, some are likely due to the mapping problem. As an example consider the tool CT1 in Table I that failed at mapping the Apache FileUpload archive to the corresponding CPE. Another source of false negatives is the accuracy of the NVD data. As it is the case for CVE-2013-2186, whenever libraries are not listed among the affected CPEs, the mapping may fail if ad hoc strategies or punctual corrections are not put in place. A third cause of false negatives is repackaging, i.e., cases in which vunerable OSS code has been extracted from the original archive and repackaged together with other code from either the application or other libraries. To the best of our knowledge, OWASP DC and the other tools we considered have problems in collecting sufficient evidences to identify one or multiple original libraries whose code was repackaged.

On the other hand, our approach fails to recognize vulnerabilities that are fixed by other means than corrections to source code. In such cases, we do lack a change list whose constructs we can find in bundled libraries, and whose execution we can trace during application tests (cf. CVE-2013-0248).

Finally, it is important to mention that, though our approach is independent from the programming language, our prototype currently supports Java whereas the other tools also support other languages.

V. Data Integration Problems

Our approach requires the integration of information stemming from different sources, e.g., vulnerability databases, VCS for managing the code base of OSS, and public OSS repositories (e.g., Maven Central[13] in case of our Java prototype). During our experiments, we found that the integration is hindered by several problems, each one requiring ad-hoc, technology-specific solutions. Such problems hamper—in general—the automation of OSS vulnerability management.

Non-uniform reporting of products affected by a vunerability.

The NVD uses the CPE standard for enumerating components affected by a vulnerability. In our experiments, we observed a non-uniform practice of assigning CPEs to vulnerabilities in OSS libraries. In some cases only the CPE of the respective library is mentioned. As an example, the affected components for CVE-2012-2098[14] are versions of Apache Commons Compress before 1.4.1 (`cpe:/a:apache:commons-compress`). In other cases also the CPEs of applications making use of the library are listed. This is the case for CVE-2014-0050, whose affected components include not only Apache FileUpload (`cpe:/a:apache:commons_fileupload`), but also several versions of Apache Tomcat (`cpe:/a:apache:tomcat`), which is just one out of many applications using this library. As mentioned in the textual description of the CVE (cf. Section III-B), JBoss Web, and other products are also affected, though, not listed as CPEs. Finally, as it is the case for CVE-2013-2186 (cf. Section IV), it may happen that only the CPEs of products making use of the vulnerable library are reported.

The above CPEs are written in the URI binding form. We made use of a subset of the supported fields, i.e., type `a` to denote an application, vendor `apache`, product `commons-compress`, `commons_fileupload`, and `tomcat`. It is important to notice that CPEs do not provide an immediate means to identify libraries.

Vulnerability and dependency management make use of different naming schemes and nomenclatures.

There exist many language-dependent technologies and nomenclatures for identifying libraries and managaging dependencies at development and build time (e.g., Maven for Java), none of which maps straight-forwardly to CPE. As an example consider the Apache HTTP Client library. Maven identifiers are of the form

```
GroupId=org.apache.httpcomponents
ArtifactId=httpclient
```

for each release as of 4.0. CVE-2012-6153 affects Apache Commons HttpClient before 4.2.3 and the CPEs listed as affected products are of the form

[13]http://search.maven.org

[14]Algorithmic complexity vulnerability in the sorting algorithms in bzip2 compressing stream (BZip2CompressorOutputStream) in Apache Commons Compress before 1.4.1 allows remote attackers to cause a denial of service (CPU consumption) via a file with many repeating inputs.

`cpe:/a:apache:commons-httpclient`

While a human can easily recognize the mapping, this is not the case for an automated solution. The problem is made even worse by the fact that Maven identifiers and CPEs may change for newer releases. As a matter of fact the syntax `commons-httpclient` used in the above CPE was also used as Maven group and artifact id for releases before 4.0.

Moreover, there exists no unambigous, language-independent way for uniquely identifying libraries once they are bundled and installed as part of an application. Our prototype computes the SHA-1 of Java archives and performs a lookup in Maven central: an ad-hoc solution that fails if the bundled library has been compiled with a different compiler than the library available in the public repository.

Vulnerabilities and VCS information of the respective patch are not linked in a systematic and machine-readable fashion. The change-list computation requires as input the URLs of VCS and commit numbers. The Patch Analyzer of our prototype uses two strategies for discovering them: *(i)* pattern matching for identifying VCS information in CVE references and *(ii)* search for CVE identifiers in VCS commit logs. While successful in case of CVE-2014-0050 and many other vulnerabilities, they still represent ad-hoc solutions depending on the discipline of developers and the quality of CVE entries. In particular, both strategies are successfull for CVE-2014-0050: *(i)* the VCS

`http://svn.apache.org/r1565143`

is listed among the CVE references; *(ii)* the commit log for revision `1565143` contains the CVE identifier:

```
Fix CVE-2014-0050. Specially crafted input
can trigger a DoS if ... This prevents the
DoS.
```

However in other cases different practices are used. As an example CVE-2012-2098 affecting Apache Commons Compress does not reference the VCS revision(s) fixing the vulnerability, nor the VCS commit log systematically references the CVE. In this example the revisions fixing the vulnerability are listed in a webpage of the Commons Compress project dedicated to security reports[15].

VI. Related Work

With the increased adoption of open-source components in commercial products, the attention to the potential risks stemming from this practice has increased correspondingly. More specifically, the problem of evaluating the impact of vulnerabilities of open-source (and more generally, of third-party) components has attracted significant attention among researchers [4], [5] and practitioners [2], [1]. Several approaches tackle the problem of vulnerability impact assessment by examining the system statically, in order to determine a measure of risk, as in [6] where the authors consider the relation between the entry points of the subject system (the potential attack surface) and the attack target (the vulnerable code). Younis et. al [3], elaborating on that idea, proposed a similar approach that also measures the ratio between damage potential and attack effort in order to estimate how motivated an attacker needs to be when targeting a particular point of the attack surface.

Our approach is complementary to these, in that our goal is to observe real executions of a system (as opposed to analyzing its structure and call graph) in order to detect whether actual executions were observed that touched a part of the code that is known to be vulnerable. Judging how likely is the exploitation of vulnerabilities that were not covered by concrete execution is not in the scope of this work (although we do plan to include that aspect in our future research).

Several tools have been proposed to help detect the use of vulnerable libraries, such as the OWASP Dependency-Check[16] or the Victims Project[17]. Both support the check of whether a project depends on libraries for which there are any known, publicly disclosed, vulnerabilities. Similarly to ours, these tools are realized as Maven plugins to minimize the barrier to adoption. They differ from our approach because their goal is to identify whether a vulnerable library is *included* in a project, whereas we concentrate on detecting whether the *vulnerable portion* of the library can be actually *executed* as part of the container project, a question that is particularly relevant for released applications.

VII. Conclusion and Future Directions

This paper presented a pragmatic approach to answer one important and time-critical question: Does a vulnerability in bundled OSS libraries affect an application? Our approach helps to assess whether urgent patching is needed in response to a vulnerability.

Up to now, we have used the current implementation of our approach to a limited set of sample projects. The evaluation done was only preliminary, but the feedback we received from the the early adopters of our tool (develoment units internal to our company) indicates clearly that the problem we are tackling is perceived as timely and extremely relevant in practice. That feedback also highlights the importance of several outstanding problems which demand further investigation. In this section we summarize the future directions of our research, which will be the topic of future works.

A. Accuracy of the Analysis

One inherent limitation of our approach, as most existing approaches to vulnerability analysis, is that it is neither sound nor complete. In particular, the reliability of our assessment is heavily dependent on the coverage achieved through executions (e.g., obtained by testing) of the application under analysis and its libraries. This has two important consequences: (A) one is related to the nature of the judgments that one can draw based on testing; (B) the other is related to the problems that

[15]http://commons.apache.org/proper/commons-compress/security.html

[16]https://www.owasp.org/index.php/OWASP_Dependency_Check

[17]https://victi.ms/, https://github.com/victims

can arise when a test suite constructed for functional testing is used as the basis of a security assessment.

Obviously, when test coverage is poor, it may happen that a vulnerability is not deemed relevant just because none of the observed executions followed a path that could reach the vulnerable library code. This says nothing about whether such a path would be feasible in practice. Furthermore, even when relying on a well-written (functional) test suite that achieves high-coverage, there might still be corner cases – not considered in functional tests – that are potentially relevant from a security standpoint and that an attacker could exploit. In fact, even a coverage of 100% of application statements does not mean that all feasible paths in the libraries are executed. At the same time, it is difficult to define an appropriate level of test coverage for libraries, since applications anyways only use a fraction of them.

Still, regarding point (A), it is important to use test suites that achieve good coverage in order to reduce the likelihood that obvious problematic execution paths go unnoticed. Regarding point (B), augmenting the functional test suite to cover corner cases and negative tests can be useful, but the above-discussed problem related to library test coverage makes us think that the problem would be tackled more effectively with some form of flow analysis. By analysing the source code of the target program together with its libraries, one could determine whether it is at all *possible* to reach vulnerable library code from the application. This method could provide very strong evidence that the vulnerable code is *unreachable*, and as such it would complement nicely our test-based method with provides very strong evidence (a proof, indeed) in the complementary case, that is when vulnerable code is indeed *reachable*. However, this approach comes with its own problems and would still need to cope with some degree of approximation. A study of the interplay of test-based analysis and flow analysis will also require to investigate which types of analyses are best suited to provide a good balance between reliability of the results and performance, especially when taking into account the the size and complexity of real commercial applications.

B. Scalability to Large Projects

While we do not have conclusive evidence nor quantitative figures to present at this time, the performance observed when using our tool on real projects is promising. We believe that the performance penalty that our tool imposes on the build process would acceptable in most practical cases. At the time of writing, we have started a systematic study of the performance of our tool, by gathering measurements taken when using the tool in large development projects with complex build structure and hundreds of dependencies.

C. Experiments to Validate the Assumptions of our Approach

The basic assumption on which this work is based (see Section I) seems sensible based on what we observed in a limited set of sample projects both in the open-source and in industrial projects. However, its rigorous validation is a prerequisite for drawing more realiable conclusions about the quality of our approach. This validation will require examining a larger number of projects and compare the results of our analysis with the actual exploitability of vulnerable libraries in the context of those projects.

D. Tackling the Data Integration Problems

As discussed in Section V, as of today the heterogeneous sources of information on which our approach relies (e.g., NVD and source code repositories) cannot be integrated reliably. One future direction is therefore to further decrease the dependency on human-supplied identifiers (e.g., Maven artifact identifiers or CPEs) and to rely only on source code as the basis for our assessment. In particular, we intend to investigate the possibility to compare method bodies of vulnerable and fixed program constructs (obtained during patch analysis) with those found during application analysis (when looking at the byte code of dependencies) in order to distinguish whether an application uses the vulnerable version of a program construct or a fixed version. This work would make it unnecessary to consider Maven identifiers or CPEs when establishing the library version used by a given application, and whether that version is among the vulnerable releases affected by the vulnerability at hand. This direction of work particularly supports the use of our tool for the assessment of old vulnerabilities, while our initial motivation was primarily the assessment of newly published vulnerabilities and fixes.

E. Continuous Integration Systems

Our approach has a very natural application in continuous build and integration systems. Here, the tool can collect data on a regular basis so that all sets S_a, S_i and T_{ai} are already populated when a new library vulnerability and a corresponding fix C_{ij} are published. This allows providing timely feedback to developers and application users, which is important for highly visible vulnerabilities such as Heartbleed. At the time of writing, we pilot the use of our tool in the build process of several SAP development groups in order to further evaluate our approach. Existing functional test suites are thereby executed twice, once without using our tool (as to avoid any impact on the fidelity of functional tests), once with our tool enabled in order to collect the required data sets.

References

[1] OWASP Foundation, "OWASP top 10 - 2013," Tech. Rep., 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

[2] Contrast Security, "The unfortunate reality of insecure libraries-reloaded," Tech. Rep., 2014. [Online]. Available: http://www1.contrastsecurity.com/the-unfortunate-reality-of-insecure-libraries

[3] A. A. Younis, Y. K. Malaiya, and I. Ray, "Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability," ser. HASE '14. IEEE Computer Society, 2014, pp. 1–8.

[4] G. Schryen, "Is open source security a myth?" *Communications of the ACM*, vol. 54, no. 5, pp. 130–140, 2011.

[5] A. Arora, R. Krishnan, R. Telang, and Y. Yang, "An empirical analysis of software vendors patching behavior: Impact of vulnerability disclosure," 2006.

[6] D. Brenneman, "Improving software security by identifying and securing paths linking attack surfaces to attack targets," McCabe Software, Tech. Rep., 2012.