CONFIRMING THE INTEGRITY AND UTILITY OF OPEN SOURCE FORENSIC TOOLS

by

Steven Morra

A Capstone Project Submitted to the Faculty of

Utica College

December 2013

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in
Cybersecurity
© Copyright 2013 by Steven Morra
All Rights Reserved
Abstract

In order to present digital evidence in court, one must prove that a series of legal standards were met when said evidence was collected or generated. In order for an open source software tool to meet these standards, the tool must be reviewed in the following ways: the tool’s source code must be reviewed and must reflect the author(s) claims as to what the tool does and no more; the tool must be compiled and sandboxed to confirm the binary’s functionality; and the tool must be stress tested to confirm consistency of output. These three steps will be accomplished during the course of this paper which will provide a general template applicable to any OSS tool. The tool will be compiled and tested inside of a sandbox to ensure that no suspicious processes are run, and no suspicious ports are opened, etc. The tool will then be tested in extreme conditions i.e. on huge, tiny, corrupted, encrypted, or illegitimate files and the resultant output will be evaluated. The goal is to equip an investigator with documentation that can be used in court to prove the legitimacy of the tool and its methods. The investigator will then learn how the tool works and when it doesn’t. This understanding as well as the documentation backing up the tool will shield the investigator from common legal attacks levied from the other side of the bench. Keywords: Cybersecurity, Professor Cynthia Gonnella, open source forensic tools, Daubert standard, PDFiD.
Acknowledgements

I’d like to thank Professor Cynthia Gonnella and Professor Leonard Popyack for helping me during the writing process of this paper. I’d also like to thank Didier Stevens, whose tool I analyzed, and who was happy to correspond with me about this paper. Mark Low was a helpful editor, and Professor Vernon McCandlish was a generous resource as well.
# Table of Contents

List of Illustrative Materials .......................................................................................... vii  
Confirming the Integrity and Utility of Open Source Forensic Tools .......................... 1  
Literature Review ......................................................................................................... 7  
Methodology .................................................................................................................. 13  
   Preliminary Considerations ....................................................................................... 13  
   Identify and Acquire a Tool, and Verify Its OSS License ...................................... 13  
   Determine a Tool's Functionality and How It Accomplishes Its Tasks ................. 14  
      Review what the author claims the tool does. ...................................................... 14  
      Document and review imported libraries. ............................................................ 14  
      Document and review each class and function. .................................................... 15  
Sandbox the Tool ......................................................................................................... 16  
Test the Tool’s Apparent Functionality ..................................................................... 17  
   Formulate test assumptions. ..................................................................................... 17  
   Test the test assumptions. ....................................................................................... 18  
Stress Test the Tool ...................................................................................................... 20  
Hash the Tool ................................................................................................................ 20  
Conclusions, Recommendations, and Best Practices ............................................. 21  
Analysis ......................................................................................................................... 22  
   Identify and Acquire a Tool, and Verify Its OSS License .................................... 22  
   Determine a Tool's Functionality and How It Accomplishes Its Tasks ............... 22  
      Review what the author claims the tool does. .................................................... 22  
      Document and review imported libraries. ............................................................ 26  
Sandbox the Tool ......................................................................................................... 27  
   Preparation. ............................................................................................................... 27  
   Manual sandboxing. ................................................................................................. 28  
   Automatic sandboxing .............................................................................................. 28  
Test the Tool’s Functionality Against the Author’s Claims and the Code’s Apparent Framework ........................................................................................................ 29  
   Formulate the test assumptions. ............................................................................. 29  
   Test the test assumptions. ...................................................................................... 30  
   Stress test the tool. ................................................................................................ 32  
Testing Conclusions, Recommendations and Best Practices ................................ 32  
Lack of Foundational Research ................................................................................. 35  
Proving Daubert Compliance .................................................................................... 36  
   OSVAT as a trusted third party. .............................................................................. 37  
Recommendations for Further Research .................................................................. 37  
   Ensuring PDFiD’s Daubert Compliance ................................................................ 37  
   Securing Closed Source’s Inherent Advantages ................................................ 38  
      Documentation. ................................................................................................... 38  
      Gaining trust. ...................................................................................................... 38  
      Daubert compliance ......................................................................................... 39  
Conclusion ................................................................................................................... 39  
References ..................................................................................................................... 41  
Appendix A .................................................................................................................... 47
List of Illustrative Materials

Figure 1 PDFiD’s sample output ................................................................. 24
Table 1 PDFiD’s Imported Libraries ............................................................. 26
Figure 2 urllib and urllib2 .................................................................. 27
Figure 3 The Windows Command Line .................................................. 29
Table 2 Assumptions and their test results ............................................. 31
Confirming the Integrity and Utility of Open Source Forensic Tools

Given today's economy, it is more important than ever for computer forensic investigators to utilize as many methods as possible to conduct investigations at lower costs. Some investigators have looked to open source tools for conducting their forensic examinations. Investigators can make use of open source software (OSS), providing them with powerful forensic tools at no cost. OSS is software that is published under an OSS license, like the GNU General Public License (GPL). The foundations of GPL include: "the freedom to use the software for any purpose, the freedom to change the software to suit your needs, the freedom to share the software with your friends and neighbors, and the freedom to share the changes you make" (Smith, 2013, p. 2). In other words, GPL licensed OSS allows anyone to view, alter, and distribute the software’s source code.

One concern that law enforcement officials might have is whether or not the evidence generated by OSS will be court admissible. In order for evidence to be admitted into court, it must comply with various evidence collection and preservation standards. Evidence collection and preservation requires the investigator to use trusted tools and methods. The purpose of this research was to document the means and methods involved in validating the integrity of open source forensic tools and their output while complying with relevant legal standards. What legal standards are applied by the courts to determine if an OSS program should be admitted? What methods and tools are necessary to prove the integrity of an OSS program has been maintained? What is the process to test and validate an OSS program? The overall process will be accomplished by breaking down the process of validating OSS into manageable steps and then to answer three questions about each step: what problem the step addresses, why the step is important, and how the step answers the problem.
Open source forensics software can be an attractive solution to anyone running a forensic lab with budget constraints. The proprietary software by AccessData, Forensic Tool Kit (FTK), can cost about $3,000 (AccessData Group, 2012). OSS tools can be obtained and used at no cost, but the traditional method of ensuring that the evidence OSS generates is admissible in court isn’t always free. For example, presenting evidence in court that was generated by an OSS tool requires an expert to testify to the tool’s efficacy. Since most forensic examiners are not necessarily programming experts, an expert witness must be called in. Expert witnesses often charge upwards of $400 an hour (Expert Witness Fees, 2004). The expert witness is required to prove that the tool complies with various legal standards, notably the Daubert standard.

In 1993, the Supreme Court articulated the Daubert standard in the case Daubert v. Merrell Dow Pharmaceuticals, Inc., 509 U.S. 579 (Brenner, 2013). The Supreme Court clarified rule 702 in the Federal Rules of Evidence ruling that a “witness qualified as an expert by knowledge, skill, experience, training, or education may testify thereto in the form of an opinion” provided the following criteria are satisfied: “the testimony is based upon sufficient facts or data; the testimony is the product of reliable principles and methods; and the witness has applied the principles and methods reliably to the facts of the case” (Hutchinson, 2012, p. 2). This paper attempts to satisfy the second and third criteria prior to legal action by proving that reviewed tools generate legitimate output, allowing the examiner to provide facts in testimony, and will demonstrate that the tool works using reliable principles and methods. This paper will be useful for anyone looking to conduct forensic examinations on a budget by serving as a template that will detail the steps required to demonstrate a piece of OSS’s general integrity and its compliance with the Daubert standard.

In order for a piece of OSS and its output to be presented as evidence in court, an expert
witness will often have to testify to the legitimacy of the OSS’s output, as well as to the soundness of the investigator’s examination methods. For example, in the case of Kumho Tire Co. v. Carmichael, the Supreme Court ruled that in order for the product of “scientific, technical, or other specialized knowledge” to be presented in court, a judge in a given case must evaluate whether or not the tool or the process that generated the evidence has been tested or peer reviewed, whether or not there is a known rate of error during evidence generation, and whether or not the process or tool is generally accepted by expert peers (Kumho Tire Co. v. Carmichael, 1999). This is also known as the Kumho-Daubert standard, which is somewhat an addendum to the Daubert standard that stipulates how a court defines scientific methodology. Other court decisions have mandated that if OSS generates evidence that is presented in court, both the prosecution and the defense must have access to a summary of that evidence as well as access to the source code of the OSS that was used to generate the evidence (Scott & Carpenter, 2004). Ensuring that a piece of OSS complies with general legal standards requires careful review.

Specifically, evidence generated by OSS that is reviewed using this paper’s template will gain further Daubert compliance in the following ways: (a) tools reviewed using methods outlined in this research will be documented as validated and tested, (b) examiners using a tool reviewed by the methods defined in this research can generate evidence that works toward satisfying the Daubert standard, and (c) this research promotes the acceptance of OSS forensic tools in general.

The process and associated documentation in this research provides anyone reviewing OSS with a template that can be applied to ensure OSS enjoys the same advantages in court as closed source software, or software whose code is proprietary. The necessity of compliance with the Daubert standard as well as closed source software’s advantages were both demonstrated in
U.S. v. Springstead, 2013, a case where a man was accused of distributing child pornography. The evidence presented in court was found by an investigator named Paul Wolpert and was generated using the proprietary forensic software suite FTK. The defense objected, claiming that Wolpert’s evidence didn’t comply with the Daubert standard but the Court of Appeals rejected this claim, citing Wolpert’s experience and his FTK certification, as well as FTK’s history in court (Brenner, 2013). The Court of Appeals judged that Wolpert didn’t have to testify on behalf of FTK’s internal workings, a judgment that may not have been made if Wolpert used OSS (Brenner, 2013).

This research documents how OSS forensic tools’ source code should be reviewed, how the tool should be sandboxed (tested in a clean and isolated environment), and how the tool should be stress tested. Ensuring that OSS code is legitimate requires the tool to reflect only what the author claims the tool does (Miller, 2005). This was confirmed using two methods: inspecting the source code, and then testing the code or its resultant binary. In order for a computer to process higher level software code, the code must be compiled into a machine-readable binary, also known as an executable (Bolton, 2013). On Windows machines, binaries are often ".exe" files, also known as programs.

Any given OSS tool will typically consist of a series of functions, and if the program is object-oriented, classes as well (Nirosh, 2011). Functions and classes handle and comprise the intermediate steps between the user’s execution of a given binary and its successful resultant output. Functions, according to Danzig (2006), are “a set of actions designed to be executed at various times” and classes are a template capable of holding data and functions (Rouse, 2005). Each intermediate step must be enumerated and evaluated, and each imported library must be documented similarly. Generally speaking, reviewing the code attempts to answer two questions
about the tool, the latter of which can be very complex: what does the tool do, and how does the tool do it?

Sandboxing involves placing an untrusted piece of software into an isolated environment and attempting to detect anomalous behavior (Rouse, 2012). A typical sandbox consists of a virtual machine that is set up to be incapable of accessing the network in order to prevent the spread of malware in the event that the untrusted software being tested is malicious. Sandboxing is difficult because the reviewer is essentially trying to prove a negative; that the tool being reviewed does not contain malware (Ghost, Hill, & Schmid, 2002). A virtual machine used to sandbox will run its own Operating System (OS), or its own “program that controls and manages the hardware and other software on a computer” (Fisher, Operating System, 2013). Sandboxing involves the careful documentation of legitimate processes running in the OS, running the binary being tested, documenting what processes are a result of the binary’s execution, and ensuring those processes are legitimate, all while containing the code as it is running (Hofman, 2013).

If the source code or the binary is clean, the tool will be tested against a series of assumptions. In 2008, the National Institute of Justice published a report that reviewed the closed source EnCase LinEn 6.01 and in the course of the report, formulated 22 different assumptions to test the tool against (Test Results for Digital Data Acquisition Tool: EnCase LinEn 6.01, 2008). Assumptions vary from tool to tool, but test similar functionality such as whether or not the tool generates accurate output, whether the tool can operate on hidden, encrypted, or corrupted files, whether the user is alerted to errors or incomplete execution, whether data is preserved in a forensically sound manner, and whether the tool can operate in a forensically sound environment. These assumptions are formulated to reflect what the author claims the tool does. Any relevant community documentation or manual pages are also considered. This can be a
difficult step because open source tools frequently lack thorough documentation and would therefore require the reviewer to look closely at the tool’s source code (Shields, 2009).

Once all of the functionality of the tool is evaluated, the tool can be stress tested. Stress tests attempt to identify what types of input might cause the tool’s output to be incomplete or inconsistent (Gheorghiu, 2005). For example, if a piece of OSS carves and then parses text files, the tester could present the tool an image of a drive containing malformed or otherwise corrupted text files. Another example of stress test criteria could be feeding the tool very small or very large files. The goal with stress testing is to attempt to get the tool to generate false positives and false negatives which would in turn help satisfy Daubert’s reliability standards (Stevens M., 2007).

Malware, or malicious software, could disqualify a piece of OSS during a case or could disrupt an entire network of computers (Roberts, 2012). Malware is the central reason this research made use of sandboxing techniques. Testing a piece of OSS in a sandbox allows for testing to prove it to be malware-free, while protecting the outside environment (Chickowski, 2013). Testing in sandboxes can be challenging because of the complex nature of modern Operating Systems as well as the inexact nature of this type of test (Ollmann, 2013). Searching for malware requires the tester attempt to prove a negative: to prove that there is no malware attached to the binary in question.

This documentation is intended to support the proposed Northeast Cyber Forensic Center (NCFC) Open Source Validation and Testing (OSVAT) center's website. The goal is to provide a resource for anyone looking to use OSS for digital forensics. In 2011, the software download repository Download.com was found to be attaching malware to the common network security open source tool called Nmap (Krebs, 2011). According to the web traffic information website
Alexa, Download.com’s parent site called CNet is the 97th globally ranked website as of 2013 (How popular is cnet.com?, 2013). Therefore, the point in time when NMap was packaged with malware, CNet was highly trusted, and Download.com was a common place people went to download software. Centralizing the validation documentation and the actual OSS code that was reviewed on the OSVAT center site in the way this research proposes alleviates the concerns examiners have when downloading OSS from unknown or untrustworthy sites where malware or adware could be attached.

Hash values will be used to prevent any entity, legal or otherwise, from questioning the integrity of any specific instance of a tool that has been reviewed and published on the OSVAT website. Calculating a file’s hash value allows a user to generate a file’s unique digital footprint (Bolton, Definition of Hash Algorithm, 2013). Once OSS forensic tools are validated, each tool’s hash value will be calculated and published on the OSVAT website along with the tool itself. That way, even if the OSVAT website’s software is compromised the way Download.com’s was, the user wouldn’t have to trust the website; the user could trust the hash value published in the software’s review. Publishing the hash value allows examiners to be certain that the OSS forensic tool in use is the same one that OSVAT reviewed. A differing hash value alerts the examiner that the integrity of the OSS forensic tool has been compromised, whether it is a damaged file or it has been altered intentionally. A matching hash value prevents either side in legal proceeding from questioning the integrity of the OSS forensic tool itself.

**Literature Review**

In 2003, Brian Carrier, author of the OSS forensic tool called The Sleuth Kit, published a paper advocating the use of OSS forensic tools using legal arguments (Carrier, 2003). In his paper, Carrier claims that the Daubert standard is easier to meet using OSS forensic tools. Carrier
explains that the Daubert standard’s stipulation that a tool or a process must be accepted by the relevant scientific community isn’t sufficiently satisfied by closed source tools. He argued that closed source tools typically point to their large user base to prove community acceptance, which Carrier judges to be less adequate compared to OSS for two reasons: the community doesn’t have access to the inner processes or the source code that constitute the closed source tool and thus can’t endorse them; and the user base for a given closed source tool typically chooses the tool for “non-procedural factors such as interface and support” (Carrier, 2003, p. 7). According to Carrier, OSS forensic tools, on the other hand, are implicitly granted community acceptance in Darwinian fashion by virtue of their propagation and their continued development and use. Carrier further argued that OSS complies better and more flexibly with Daubert standards relating to publication, peer review, and testing (2013).

In 2012, law professor Susan Brenner recounted a case where a closed source tool developed by the FBI was subject to a legal challenge under the Daubert standard; a case which challenges many of Carrier’s claims above. The FBI tool in question, called enhanced peer-to-peer software (EP2P), is an altered version of the popular file sharing tool known as LimeWire (U.S. v. CHIARADIO, 2012). LimeWire is a P2P file sharing tool, meaning it can allow two users to share a file without a third party, hence peer to peer (P2P). The difference between EP2P and LimeWire is that EP2P is capable of identifying the source of a given file using data such as the file sharers’ IP address and their local Internet Service Provider, while satisfying various legal and forensic best practices (Brenner, 2012). Brenner’s recounting of the case suggests that the fact that the tool was closed source and that EP2P’s source code isn’t available to either the public or the agents who operate the software shielded the code from review. Still, the Daubert standard had to be satisfied, which the FBI accomplished by submitting testimony from FBI
agent Michael Gordon who had participated in over eighty investigations involving EP2P. Gordon stated that he had been involved in testing the program, claimed that in his experience, EP2P has never yielded a false positive, and he demonstrated a method of confirming EP2P’s ability to trace a given file’s transfer to its genesis (Brenner, 2012).

Brenner refers to U.S. v. Springstead, 2013, where Springstead was suspected of crimes and an investigator named Wolpert examined his computer using FTK, a closed source forensic tool (Brenner, 2013). Wolpert found evidence of child pornography which was then admitted to court. Wolpert testified about his qualifications as a digital forensics expert, stating that he had extensive experience using FTK including a certification issued by FTK’s parent company, AccessData. He also had visited semiannual training events, and he described a detailed methodology he followed to generate the evidence he submitted, further reflecting an understanding of how to use FTK. Springstead’s defense levied a series of claims against Wolpert. According to Brenner, the claims were all true, including that Wolpert:

- did not know what the error rate on the software was, if any, did not know how the software was designed, did not know how the software purported to obtain and sort the information from the hard drive, and indicated that his work was not peer reviewed. (Brenner, 2013, para. 10)

The defense further stated that:

- Wolpert could not explain significant anomalies like how images could allegedly be on the hard drive before it was created and how certain files could have a date of transfer that was before the date the file was created, per the report that FTK produced. (Brenner, 2013, para. 12)

In the end, the Court of Appeals ruled that Wolpert’s testimony and evidence were sound, and
that Daubert’s reliability tests are ‘flexible’ (Brenner, 2013).

The unimpeachability of evidence generated by closed source tools in the U.S. v. Springstead is remarkable. However, in a paper written for the Virginia Journal of Law and Technology, Erin E. Kenneally paints a much bleaker picture for closed source tools. Kenneally argues that the Daubert standard can be circumvented because closed source tools’ output can be presumed to be reliable for inadequate or incomplete reasons. Kenneally explains that courts have traditionally assumed that closed software cannot be manipulated, which she contrasts with the fact that closed source, “operators proceed on blind faith that the software will perform as advertised despite constant bug fixes, patches, hacks, and vulnerability alerts to correct flaws” (Kenneally, 2001, p. 75).

Additionally, Kenneally takes issue with the idea that market share is a valid metric. In her view, courts have presumed that widespread closed source tools must be reliable by virtue of being widespread, without acknowledging, “the pervasiveness of unreliable software,” that, “has molded a community of users that has grown increasingly desensitized to unreliable software and its accompanying contraindications as an accepted cost and defining attribute of networked society” (Kenneally, 2001, p. 79). Kenneally explains further that market share is a proper metric when questioning the legitimacy of a calculator's output, an assumption that has guided rulings on closed source software. In Kenneally’s view, the fact that data is “subject to a panoply of algorithms” when processed by modern computer software, doubt rather than certainty should be cast on closed source software because closed source hides its algorithms (Kenneally, 2001, p. 79).

Finally, Kenneally believes that proving closed source tools produce reliable output is impractical. She proposes a hypothetical case involving logs generated from the closed source
Microsoft IIS. In order for a litigant to question or confirm the legitimacy of IIS’ output, the litigant, “may very well have to expend great resources to establish, through documentation and testimony concerning product development, management, testing, and implementation, reliability sufficient to meet this threshold” (Kenneally, 2001, p. 90). This includes access to the closed source code; the disclosure of which, Kenneally deems unlikely. As an alternative that addresses all of the shortcomings of closed source software articulated above, Kenneally proposes OSS.

Not only does Kenneally describe OSS as an inevitable wave of the future, she believes it to be what she phrases, “the digital Daubert” (Kenneally, 2001, p. 114). By "the digital Daubert" she means that OSS forensic tools can prove that they are peer reviewed, published, falsifiable, generally accepted, and that their rate of error has been established (Kenneally, 2001, p. 82). Kenneally implies that any advantages gained by using closed source forensic software hoping to comply with the Daubert standard is similar to security advantages gained by the idea of security by obscurity. At some point, the closed source code can be known and will be questioned and therefore it is better to publish and evaluate the source code openly.

The National Institute of Science and Technology (NIST) runs a Computer Forensic Tool Testing (CFTT) website that periodically releases assessments of forensic software tools by comparing each tool against a series of assertions, and publishing their findings. NIST published a report in May of 2013 for the National Institute of Justice (NIJ) detailing tests on the FTK Imager command line tool (Test Results for Digital Data Acquisition Tool: FTK Imager CLI 2.9.0_Debian, 2013). The report doesn’t unequivocally state whether or not the source code was reviewed, which could be assumed to have been impossible because of FTK’s proprietary nature. Another detail that is omitted from the FTK Imager report is a comprehensive list of the commands and overall methods used to generate a given result. NIST’s CFTT does not appear to
review OSS tools aside from the ubiquitous Linux command ‘dd’ and tools with functionality similar to ‘dd’ (Test Results for Disk Imaging Tools: dd Provided with FreeBSD 4.4, 2004).

According to OSS forensic tool author Cory Altheide, conducting “reproducible tests” “performed at regular intervals” against proprietary tools is sufficient to satisfy the Daubert Standard (Altheide & Miller, Validating Proprietary Digital Forensic Tools: A Case for Open Source, 2011). However, he later concedes that the source code of a tool could still be questioned, in which case an expert would be required to testify to the legitimacy of the tool’s code. Citing the Casey Anthony trial, Altheide explains the importance of documenting and acknowledging potential false positives. In the trial, two proprietary tools produced conflicting results on identical data, allowing the defense to question the integrity of the tools and the evidence (Altheide & Miller, 2011).

Returning to Carrier, “to further the acceptance of analysis tools in a legal setting, the following steps must be taken in the future:

- Development of comprehensive tests for file system (and other) analysis tools in addition to the ones that NIST has already developed for disk imaging tools
- Publication of tool design to help create more effective tests
- Creation of a standard for calculating error rates for both tools and specific procedures
- Publication of specific procedures that a tool uses. While open source tools already publish their source code, they should also describe the procedure in words.
- Public debate on the published procedural details to ensure that they are agreed upon.” (Carrier, 2003, p. 9)

Each of Carrier’s bullet points are addressed throughout this body of research.
Methodology

Preliminary Considerations

The goal of conducting a software review is to help preemptively satisfy the Daubert standard. This is accomplished by documenting the methods and results of each step herein after, and then authoring a formalized report consisting of data gathered while conducting the review. The formal report is expected to be posted to the OSVAT website. Due to the lack of foundational research found on this topic—which is explained in the concluding section—parts of this methodology section are adapted from CFTT’s closed source forensic tool testing methodology, and are focused on satisfying the Daubert standard (CFTT Methodology Overview, 2003).

Identify and Acquire a Tool, and Verify Its OSS License

The first step in validating a tool is to identify an OSS tool to validate. Relevant criteria when selecting a tool includes finding a tool already in use, identifying a tool that hasn’t been reviewed, and engaging the digital forensics community in places such as Forensics Wiki or NIST’s CFTT website in the hopes of finding a tool with vital or unique capabilities (Garfinkle, 2013). The selected OSS tool must be confirmed to actually be open source and published under an open source license. Typically, a tool will display its OSS license in its own code, in a readme file that comes packaged with the tool, or its license will be posted alongside its source code in a centralized repository website like Google’s official developer site (code.google.com), or the popular project hosing site known as GitHub (github.com). This is an important step because it can be impossible or illegal to gain access to, review, or edit proprietary or closed source code (Holst, n.d.).
Determine a Tool's Functionality and How It Accomplishes Its Tasks

Determining the functions of a tool and how it accomplishes its tasks can be difficult and time consuming. For example, a Python tool called Volatility consists of over five thousand lines of code (Walters, 2008). Reviewing each line of Volatility's code requires a sufficient level of Python proficiency and the time required may be excessive. A line by line review is often unreasonable and isn’t necessary to demonstrate compliance with the Daubert standard. Altheide was quoted in the literature review stating that “reproducible tests” “performed at regular intervals” was sufficient to demonstrate Daubert compliance (Altheide & Miller, Validating Proprietary Digital Forensic Tools: A Case for Open Source, 2011). This step will articulate a level of granularity at which source code should be reviewed. Code should be reviewed in a way that a reviewer will find manageable while satisfying the Daubert standard.

Review what the author claims the tool does. Once an OSS tool has been identified, the reviewer should have a general feel for why the digital forensic community would want it validated and what the tool is purported to do. In order to clarify and focus these claims, manual pages and community documentation should be collected and studied to write a summary of each high level function the tool is purported to be capable. Eventually, the tool's alleged capabilities will be compared with what the tool actually does. This preliminary step allows the reviewer to place libraries and functions into context and build assumptions that will be tested against the tool later on, and reflects and repurposes a few steps CFTT recommends for testing tools, namely that “NIST reviews the tool documentation” (CFTT Methodology Overview, 2. Tool test process, 2003).

Document and review imported libraries. Programs typically begin by importing standard libraries capable of common tasks, facilitating the reuse of code. “A software library is
a suite of data and programming code that is used to develop software programs and applications” (Janssen, Definition - What does Software Library mean?, 2013). Libraries are important to review because they facilitate the use of code existing outside the OSS tool’s code. Additionally, examining each library will allow the reviewer to put functions’ constituent lines of code into context. The reviewer should document and briefly describe each library. Reviewers can briefly summarize commonly used libraries, but obscure libraries should be scrutinized and described thoroughly (Ligh, Adair, Hartstein, & Richard, 2011, p. 603).

**Document and review each class and function.** Functions and classes constitute the framework and the intermediate steps executed by a piece of software to achieve its overall functionality (Price, 2005). Reviewing these intermediate steps seeks to satisfy the Daubert standard by showing that the tool has been subject to maintenance standards and operational controls, and that the tool has been subject to peer review. Additionally, reviewing each class, library, and function allows the reviewer to understand a tool’s functionality and how it accomplishes its tasks (Miller, 2005). This understanding guides steps relating to testing later on in the review process, and documents the legitimacy of the tool and its code at a granular level.

Reviewing classes and functions can be done manually, but automated tools are useful. Tools like David Wheeler’s Flawfinder and Brian Chess’ RATS (Rough Auditing Tool for Security) can be used to find vulnerabilities in code (Vadalasetty, 2003). These tools search for and flag sequences of code that pose a potential security risk. Visustin is another helpful tool that organizes a tool’s code into a flowchart, helping visualize the code review process (Visustin v7 Flow chart generator, 2013). The purpose of this step is to explicitly state each high level function of a given tool because, as security expert Dan Geer put it, “only when you know everything that *did* happen with your data can you say what did *not* happen with your data”
Sandbox the Tool

Computer science expert Margaret Rouse defines a sandbox as, “an isolated computing environment used by software developers to test new programming code” (sandbox, 2005). Sandboxing helps satisfy the Daubert standard by proving that the tool being reviewed is reliable, the tool’s error rate has been established, and it shows that the tool has been subject to empirical testing (Mahle, 1999). Referring specifically to the ThreatExpert sandboxing website, the book titled Malware Analyst’s Cookbook explains that, “sandboxes record changes to the file system, registry keys, and incoming/outgoing network traffic, then make the results available to you in a standardized report format” (Ligh, Adair, Hartstein, & Richard, 2011, p. 100).

Zero Wine is a helpful tool that can test Windows executables on many Linux platforms or Mac OSX, provided the Windows emulation software known as WINE is installed alongside it (Koret, 2013). Zero Wine tracks the behavior of any executable it is given. Another method of sandBoxing uses virtualization software like the free and open source Virtual Box (Agarwal, 2009). Virtual Box creates a virtualization layer that mimics the hardware of a computer, or a virtual machine (VM) (Rouse, 2011). VMs are helpful when sandboxing because they are segmented from the rest of the computer; if a piece of malware infects the VM’s OS (client), the original OS (host) running the virtualization software remains unaffected. A simple sandbox could consist of Virtual Box running an Ubuntu Linux distribution. Virtual Box could be configured to disable the VM’s network connections, and Nmap, a tool capable of detecting attempted network connections, could be running to see if any processes attempt to reach out to the Internet or attempt to open ports (Brockmeier, 2010). As an executable is run, the analyst would determine which processes were being run as a result of the executable being sandboxed,
and which processes constituted normal OS behavior.

The *Malware Analyst’s Cookbook* articulates seven steps that constitute a complete sandbox review, “(1) Revert/re-image the target, (2) Copy or transfer malware, (3) Pre-execution tasks, (4) Execute malware, (5) Post-execution tasks, (6) Acquire and analyze RAM and (7) Analyze the hard drive” (Ligh, Adair, Hartstein, & Richard, 2011, p. 240). This process assumes a given executable is malware-laden, but, for the purposes of this research, steps 2 and 4 could be amended to copying or executing the tool under review. However, these seven steps require extensive testing, which is why reviewer is encouraged make use of either Zero Wine or one of the sandboxing tools the *Malware Analyst’s Cookbook* recommends. Each of these tools are listed in Appendix B (Ligh, Adair, Hartstein, & Richard, 2011). Additionally, the software tool Sandboxie was used in this testing process. Sandboxie is automated, user friendly, and useful for testing OSS (Ravenscraft, 2013).

**Test the Tool’s Apparent Functionality**

Once the reviewer has an overview of what the tool does, has an understanding of how the tool operates, and is confident in the tool’s code, a series of assumptions can be derived from the overview. Assumptions should reflect the purported and the apparent functionality of the tool. Those assumptions can then be tested and evaluated. This section will satisfy the Daubert standard by further demonstrating peer review, by showing that the technique is falsifiable and testable, by demonstrating the tool’s reliability, by defining and documenting operational controls and limits, and by documenting a known or potential error rate.

**Formulate test assumptions.** Test assumptions synthesize both the author’s claim about the tool’s functionality and the apparent functionality of the tool. If possible, test assumptions should result in binary answers, meaning the tool passes a test assumption, or it fails it. Test
assumptions should be concise and clearly defined. The properties of assumptions outlined in this section were defined to reflect CFTT’s general testing methodology (CFTT Methodology Overview, 2003).

Assume that a tool is believed to detect JavaScript inside a Portable Document Format (PDF) file (an Adobe Acrobat file format is useful for presenting users “magazine articles, product brochures, or flyers”) (Vance, 2010, para. 1). A sound test assumption could be the tool accurately detects the existence of JavaScript in a version 1.5 PDF file. When tests are conducted to measure this assumption, the reviewer should document exactly what machine, OS, application version, PDF file version, and any other environment-specific variables that were used in each test. These details should be documented either in an appendix or in the relevant assumption. This particular assumption would be helpful to test because a PDF embedded with JavaScript is a typical method in which malware is spread, an insight that should be gleaned when the tool’s documentation and the forensic community is consulted (Yonts, 2010).

Earlier in the process of reviewing a tool capable of analyzing PDF files, the reviewer would learn that malware is sometimes embedded into PDF files using JavaScript and code obfuscation (Robertson, 2012). Code obfuscation is a technique used to make it difficult for a person or a security program to detect what a given chunk of code actually does (Lyashko, 2012). Therefore, fully reviewing this tool requires testing the tool to see if it can detect obfuscated JavaScript as well. Therefore, the reviewer should amend the example test assumption to the tool accurately detects the existence of un-obfuscated JavaScript in a version 1.5 PDF file, and then the reviewer should add a second assumption that tests the detection of obfuscated code.

**Test the test assumptions.** Once the reviewer has a set of test assumptions on hand,
testing can begin. When testing the Linux imaging tool called dd, CFTT proposed the execution of three steps: to prepare the destination drive, to run dd, and then to measure the results using hash values (Setup and Test Procedures, 2002, p. 10). Similarly, testing the assumption that the tool accurately detects the existence of un-obfuscated JavaScript in a version 1.5 PDF file requires a few steps. The first step would be to test the tool against PDF files that contain JavaScript, ensuring that the tool doesn’t throw a false negative. Then, testing the tool against PDFs without JavaScript will ensure that the tool doesn’t throw false positives. Testing for both false positives and false negatives is similar to the final conformation step CFTT used to test dd (Setup and Test Procedures, 2002).

One simple, if blunt, method of stress testing is known as fuzz testing or fuzzing. Fuzzing is typically used to test an application for security vulnerabilities by presenting it with huge amounts of random data in an attempt to make the application crash or otherwise malfunction (Rouse, 2010). The idea behind fuzzing is that at some point some set of code or data could make an application unstable, and that feeding the application random input will eventually find a sequence the application has difficulty processing (Dormann, 2010). However, satisfying Daubert doesn’t require that a tool’s stability is proven, rather, the legitimacy of a tool’s internal processes and the consistency of its output is what must be shown. Therefore, working from a previous example, feeding a PDF parsing tool a PDF file with legitimate headers and footers that contains huge amounts of random data, none of which would have any actual JavaScript, could be used to detect false positives (Dorn, et al., 2009). Note: a file’s headers and footers, also known as file signatures or magic numbers, are conventions that files follow to connote what type of file they are. They’re typically between two and eight bytes long and are called headers and footers because they often append and/or prepend the file (Nadel, 2013).
The assumption the tool accurately detects the existence of un-obfuscated JavaScript in a version 1.5 PDF file, when tested, results in a binary answer. Even if this is the case, the tool can exhibit anomalous behavior. Programs can crash or otherwise malfunction. When this occurs, the anomalous behavior should be documented, the assumption that caused the anomaly must be more thoroughly tested, and the reviewer should attempt to find what caused the anomaly (SWGDE Recommended Guidelines for Validation Testing, 2009). Anomalous behavior is described in a CFTT forensic test report and is called a variance (Gavrila & Fong, 2004).

**Stress Test the Tool**

The Daubert standard stipulates that a tool’s known or potential error rate must be understood and documented (Brenner, 2013). Ssdeep, a tool capable of determining whether or not two pieces of suspected malware are similar, can be tricked into throwing a false negative if a file being compared is sufficiently appended with enough benign data (French, 2011). In order for ssdeep to comply with Daubert, its error rate must be established in advance, and the investigator using the tool must understand when ssdeep functions soundly, and when it doesn’t. The goal of this section is to use the understanding gained from examining the classes and functions that make up the tool to get the tool to throw a false positive, a false negative, or otherwise malfunction, a goal which attempts to reconcile the gap between the Daubert standard and the current state of tool testing.

**Hash the Tool**

Once an OSS tool has been successfully reviewed, a hash value should be calculated on the tool’s source code or its resulting binary (executable). Hash values are the product of a cryptographic algorithm that has various uses in computer science. Hash values are commonly used in digital forensics to ensure the integrity of a file. Specifically, SHA-1 calculates hash
values on files (or OSS tools) and has a positive history in court in rulings such as United States v. Glassgow, 2012. In the case, an expert testified that Glassgow possessed a video clip that had the same SHA-1 hash value as a file containing child pornography and that therefore there was a 99.9999% probability that the two files matched (Federal Evidence Review, 2012). If even one bit of a file is changed, its resulting hash value drastically changes (Northcutt, 2008).

Ensuring that the tool used by an examiner is the same tool that gets reviewed is accomplished by the following steps: an OSS tool is validated and its associated SHA-1 hash value is calculated; both the tool’s source code and its hash value are published on the OSVAT center's website; an investigator then downloads the source code of the validated tool, calculates the hash value of what was downloaded, and compares the calculated hash value with the published hash value. This is an important step because it allows the examiner to prove the integrity of the code that was validated and tested matches exactly the code they downloaded (Sudharsan, 2011).

Conclusions, Recommendations, and Best Practices

After a careful review of the tool, the findings should be published in a formal document to be posted on the OSVAT website. In addition to the methods and results of the testing, the OSVAT document should contain a results summary section. The results summary section should be written for programming laypeople, should summarize the confirmed functionality of the tool, and should summarize all of the tool’s shortcomings or situations that caused the tool to fail. The Scientific Working Group on Digital Evidence (SWGDE) authored a series of recommendations and a template that can be used as a guideline when authoring OSVAT documents (SWGDE Recommended Guidelines for Validation Testing, 2009). Sometimes OSS tools lack documentation. When this is the case, the report should contain an optional manual.
page that details each function the tool is capable of.

Analysis

Identify and Acquire a Tool, and Verify Its OSS License

Didier Stevens’ PDFiD will be used to demonstrate how an OSS tool should be reviewed (Stevens, 2008). Didier Stevens has authored several PDF forensic tools and PDFiD is a logical tool to analyze first because PDFiD attempts to detect whether or not a given PDF is potentially malicious. Additionally, PDFiD can strip a PDF of malicious strings. Stevens suggests that, “the idea is to use this tool first to triage PDF documents, and then analyze the suspicious ones with my pdf-parser” (Stevens, pdfid.py, 2008). In other words, PDFiD can look at a series of PDF files, can identify the potentially malicious ones, and then can strip the potentially malicious PDFs of their malicious contents. Once the PDF has been neutralized, an investigator can then use pdf-parser to further investigate the benign files.

Another reason PDFiD was selected was because of its widespread use and appeal. In 2011, PDFiD was included in BackTrack 5, a penetration testing and forensics Linux distribution (Stevens, 2011). PDFiD’s source code is published on Stevens’ website, and code in version 0.1.2 of the tool states that its source code was put in the public domain without copyright (PDFiD, 2013).

Determine a Tool's Functionality and How It Accomplishes Its Tasks

Review what the author claims the tool does. According to Stevens’ website, PDFiD will scan a PDF document and return a count of the following strings, “obj endobj stream endstream xref trailer startxref /Page /Encrypt /ObjStm /JS /Javascript /AA /OpenAction /AcroForm /JBIG2Decode /RichMedia /Launch /EmbeddedFile /XFA” (pdfid.py, 2008). Stevens explains that the strings, “/JS” and “/JavaScript” are specified because nearly all of the malicious
PDF documents he’s found actively infecting machines use JavaScript to “exploit a JavaScript vulnerability and/or to execute a heap spray” (pdfid.py, 2008). The tool searches for, “/AA”, “Launch” and “/OpenAction” because these strings can be used to launch JavaScript when the PDF is opened. Finally, Stevens states that PDFiD is capable of handling code obfuscation, and that the strings quoted above will be detected even if they’re obfuscated (Stevens, 2008).

Didier Stevens’ blog describes the following circumstances which cause PDFiD to throw false positives, “sometimes PDFiD will give you false positives for /JS and /AA. This happens with files a couple of MBs or bigger, because it’s statistically very likely that /AA or /JS (only three bytes long) appear inside a stream” (PDFiD: False Positives, 2013). When Stevens talks about /JS and /AA appearing in a stream, he is referring to an aspect of PDF syntax that PDF authors sometimes use to encode, obfuscate, or encrypt parts of their code. This means that if the byte sequence /AA appears in an encrypted stream, PDFiD will detect /AA, but the PDF file won’t interpret the sequence as the command /AA. Rather, /AA will get decrypted, and the bytes that the PDF eventually interprets will be changed (Stevens, 2008). Figure 1, taken from Stevens’ website, demonstrates some of the basic functionality of PDFiD, namely that it is a command line tool that takes one PDF file as input and displays a count of each instance of a given string, with any obfuscated instances in parenthesis (Stevens, 2008).
Figure 1. PDFiD’s sample output. The syntax PDFiD accepts in a Windows machine, provided Python is installed, is `pdfid.py filename`. The first time PDFiD was executed using an exe file as input, causing PDFiD to fail. The second time PDFiD was executed was successful. The third time PDFiD was executed shows how the tool displays code obfuscation: using parenthesis.

Additionally, PDFiD displays a few options describing some functionality when it gets run with the help flag (-h). According to Stevens’ blog, PDFiD can scan a given directory for PDFs, it can “display all the names” with the –a flag, it can “display extra data, like dates”, it can be forced to scan a files lacking the proper PDF header, and it can disable JavaScript and auto launch (PDFiD, 2013). Forming and then testing assumptions requires focus and detail, detail which isn’t always reflected in a tool’s documentation, as is the case with PDFiD. When Stevens claims that PDFiD can “display all the names” what he is actually referring to is a subset of pre-defined functions called "names" that have been embedded into PDF files since version 1.1 (PDFiD, 2013). For example, one name that PDFiD looks for is “OpenAction” which, according
to Adobe’s 2008 PDF specifications manual, "OpenAction" is a name that is used to specify “a destination that shall be displayed or an action that shall be performed when the document is opened” (Document management — Portable document format — Part 1: PDF 1.7, 2008, p. 74). Therefore, displaying all the names would show a list of each embedded PDF-specific function and presumably a count of their occurrences. Stevens’ claim that PDFiD can “display extra data, like dates,” will require a test or close inspection of the code before a focused assumption can be crafted around it (PDFiD, 2013).

Additionally, Didier Stevens claims that PDFiD is capable of handling malformed PDF documents (Stevens, 2009). Malformed PDF files are typically normal or popular PDF files that have been appended with malicious code (Gavin, 2002). One interesting case from early 2013 occurred when the cybersecurity firm Mandiant published an extensive and popular report purporting to show instances of Chinese espionage occurring on the Internet. Only a few days after the report was published, a malware-laden version of the report was found in the wild that used the original Mandiant report in PDF form with malicious data appended to it, so as not to arouse suspicion and to facilitate malware’s spread (Raff, 2013).

Malformed PDFs have some common characteristics that aren’t present in normal PDF files. According to Stevens, one can expect “the entropy of bytes inside stream objects to be close to the maximum value of 8.0” and further explained that outside of stream objects, entropy is closer to 4.0 or 5.0 (Malformed PDF Documents, 2009). Stevens further explains that PDFs could contain multiple instances of the string “%%EOF”, connoting that the file has been incrementally updated, and that the final instance of %%EOF doesn’t typically have any data after it (PDFiD, 2013). To understand what Stevens is talking about, it is important to understand PDF stream objects and entropy. Recall that a stream, or stream object, is a part of a PDF file
that its author wanted to encode, compress, encrypt, or obfuscate. Entropy is a measure of randomness and an indication that data has been encrypted or compressed (Heylighen & Joslyn, 2001).

To summarize Stevens, data within a stream object can be expected to have high entropy whereas data outside of a stream object isn’t expected to have high entropy. However, malformed PDFs can sometimes have data that’s high in entropy outside of object streams, typically appended to the PDF, an example of which is shown in Appendix A in Figure 5 (Stevens, 2009). PDFiD is apparently capable of measuring entropy inside and outside of stream objects, and is capable of measuring the amount of data has been appended to a PDF in order to detect whether or not a given PDF file is potentially malformed (Stevens, 2009).

**Document and review imported libraries.** Table 1 contains each library that is imported at the beginning of pdfid.py along with their respective use.

Table 1

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Functionality</th>
<th>Common?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optparse</td>
<td>The optparse library facilitates the use of arguments.</td>
<td>Y</td>
</tr>
<tr>
<td>Os</td>
<td>The os library gives the tool OS dependent functionality.</td>
<td>Y</td>
</tr>
<tr>
<td>Re</td>
<td>The re module provides regular expression matching operations.</td>
<td>Y</td>
</tr>
<tr>
<td>xml.dom.minidom</td>
<td>The xml.dom.minidom allows the tool to parse and manipulate XML files in various ways.</td>
<td>Y</td>
</tr>
<tr>
<td>traceback</td>
<td>The traceback library allows the tool to handle errors and exceptions.</td>
<td>Y</td>
</tr>
<tr>
<td>Math</td>
<td>The math library contains many mathematical functions.</td>
<td>Y</td>
</tr>
<tr>
<td>Operator</td>
<td>The operator library allows the tool to translate standard operators into functions.</td>
<td>Y</td>
</tr>
<tr>
<td>os.path</td>
<td>The os.path library allows the tool to manipulate paths.</td>
<td>Y</td>
</tr>
<tr>
<td>Sys</td>
<td>The sys library allows the tool to access system-specific parameters and functions.</td>
<td>Y</td>
</tr>
<tr>
<td>Json</td>
<td>The json library allows the tool to encode and decode in the JSON format and is helpful when dealing with JavaScript.</td>
<td>Y</td>
</tr>
<tr>
<td>zipfile</td>
<td>The zipfile library allows the tool to handle zip files.</td>
<td>Y</td>
</tr>
<tr>
<td>urllib2/urllib</td>
<td>The urllib library defines functions and classes capable of opening URLs and other similar tasks.</td>
<td>Y</td>
</tr>
</tbody>
</table>

*Note. Each of the descriptions under the functionality section that aren’t specifically cited was taken from the Python Module Index documentation pages (Python Module Index, 2013).*

26
The community documentation, the authors’ claims about his tool, and the manual pages have suggested no functionality relating to accessing the Internet. Therefore, the fact that the libraries urllib and urllib2 are capable of reaching out to the Internet makes them suspicious and requires further inspection. Figure 2 below proves that urllib/urllib2 libraries are imported for benign reasons, and that there was a discrepancy between PDFiD’s code and its purported functionality, likely due to insufficient and incomplete documentation.

```python
try:
    import urllib2
    urllib23 = urllib2
except:
    import urllib.request
    urllib23 = urllib.request

# Convert 2 Bytes If Python 3

def CBytes(string):

class cBinaryFile:
    def __init__(self, file):
        self.file = file
        if file == '':
            self.infile = sys.stdin
        elif file.lower().startswith('http://') or file.lower().startswith('https://'):
            try:
                if sys.hexversion >= 0x020401F0:
                    self.infile = urllib23.urlopen(file, timeout=5)
                else:
                    self.infile = urllib23.urlopen(file)
            except urllib23.HTTPError:
                print('
Error accessing URL %s' % file)
                print(sys.exc_info()[1])
                sys.exit()
```

*Figure 2.* urllib and urllib2. This shows the only instances in which urllib and urllib2 appear in PDFiD’s source code. It appears that depending on the version of Python being used to run the tool, either urllib or urllib2 is imported. This code snippet appears to show that the tool is capable of taking a PDF as input from a URL over HTTP or HTTPS, a function which is benign but was undocumented.

**Sandbox the Tool**

**Preparation.** Three different types of PDF files were prepared and used during the course of this review: benign, manual, and infected. The benign PDF is a version 1.5 PDF that was created using a draft of this paper, and was converted from a docx file to a PDF file using Microsoft Word 2010. The manual PDF was downloaded and repurposed from a posting made
on Didier Stevens’ website that discussed the structure of PDF files and provided an example called, “Hello World.pdf” (Quickpost: About the Physical and Logical Structure of PDF Files, 2008). The manual PDF is a version 1.1 PDF and is described as manual because, in order to test instances of code obfuscation and the accurate detection of each and every string, the code was edited manually, using Notepad ++; a tool used to edit source code that works similarly to Windows’ native notepad program (Ho, 2011). The infected PDF was obtained from Utica College’s ECJS website and contains malicious JavaScript. The infected PDF is version 1.1, and is named pdf1.pdf. These three files were prepared and used to conduct the sandboxing tests.

**Manual sandboxing.** PDFiD was placed into a VM running a clean install of Ubuntu 12.10 32-bit inside of VMWare Workstation 9.0.2. The VM was isolated from the LAN and therefore the Internet as well. The only currently open and listening port was 631; an enabled port by default that VMWare uses for printing. The benign and the manual PDF files were zipped, dropped into the VM, and then unzipped. PDFiD ran successfully on both files, and the Linux shell commands `ss –l` and the `netstat –tulpn` commands reflected no change in attempted network activity. To clarify, `ss` and `netstat` are basic Linux shell commands that list running processes and active network connections, respectively. This manual approach is blunt and not exhaustive. Therefore, it is not optimal for a small and focused tool like PDFiD, but was included in this research because it will be applicable to other tools, and serves as an introductory demonstration of manual sandboxing. Automatic sandboxing of the kind described in the methodology section would be more applicable and comprehensive.

**Automatic sandboxing.** PDFiD was then sandboxed using Sandboxie. Sandboxie is an automated sandboxing tool, which simplifies the sandboxing process and documents the results more clearly. A Windows 7 64-bit machine was used to run an instance of Windows Command
Line within an instance of Sandboxie. Sandboxie version 4.06 64-bit was configured to prevent and alert the user to any attempted network connections. The results of the test are shown in Figure 3. No network connections were detected, and the only program that PDFiD accessed was Python. PDFiD passed its sandboxing test.

![Sandboxie Interface](image)

Figure 3. The Windows Command Line. The Windows Command Line (cmd.exe) was opened in Sandboxie and was identified with a Product ID (PID) of 8168. As expected, using PDFiD on a folder containing the benign and the manual PDF files caused the tool to run python.exe and nothing else. Note: the window on the right is the Windows Command Line showing the command that was run, and the window on the left is Sandboxie showing that the only open processes were cmd.exe (the Windows Command Line) and python.exe.

**Test the Tool’s Functionality Against the Author’s Claims and the Code’s Apparent Framework**

**Formulate the test assumptions.** Each assumption was derived from the step, “determine a tool's functionality and how it accomplishes its tasks”.

2. PDFiD accurately counts each instance, even when the strings are obfuscated.
3. PDFiD is capable of scanning a directory for PDFs.
4. PDFiD successfully displays each instance of each PDF name.
5. PDFiD can accurately measure a PDF file’s entropy inside and outside its streams.
6. PDFiD successfully scans a file lacking the proper PDF header.
7. PDFiD successfully disables JavaScript and auto launch.

8. PDFiD successfully takes a URL as input.

9. PDFiD does not throw a statistically significant false positive on 5, 10, and 50 MB of random data.

**Test the test assumptions.** Each test assumption was tested on the Windows Command Line, on a Windows 7 64-bit OS. Some of the results were confirmed using Notepad ++. Each assumption and the results of its test are detailed in Table 2.
## Table 2

### Assumptions and their test results

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Command</th>
<th>PDF files / Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PDFiD accurately counts every instance of each string detailed in the “Formulate the test assumptions” step.</td>
<td><code>pdfid.py [filename]</code></td>
<td>manual, benign, malicious Each string was tested, and the findings and its confirmation are detailed in Appendix A.</td>
<td>Pass</td>
</tr>
<tr>
<td>2. PDFiD accurately counts each instance, even when the strings are obfuscated.</td>
<td><code>pdfid.py [filename]</code></td>
<td>manual</td>
<td>Name were obfuscated using hex encoding, and strings were obfuscated using newline escaping, octal encoding, hex encoding, and hex whitespacing.</td>
</tr>
<tr>
<td>3. PDFiD is capable of scanning a directory of PDFs.</td>
<td><code>pdfid.py –s [directory]</code></td>
<td>manual, benign</td>
<td>This command successfully runs pdfid.py on each PDF in the specified directory.</td>
</tr>
<tr>
<td>4. PDFiD successfully displays each instance of each PDF name.</td>
<td><code>pdfid.py –a [filename]</code></td>
<td>manual, benign, malicious Each object and command name is displayed.</td>
<td>Pass</td>
</tr>
<tr>
<td>5. PDFiD can accurately measure a PDF file’s entropy inside and outside its streams.</td>
<td><code>pdfid.py –e [filename]</code></td>
<td>manual, benign, malicious</td>
<td>This command checks the header, footer, and computes the file’s entropy, both inside and outside the PDF’s streams.</td>
</tr>
<tr>
<td>6. PDFiD successfully scans a file lacking the proper PDF header.</td>
<td><code>pdfid.py –f [filename]</code></td>
<td>random data file [see Appendix A for details]</td>
<td>Pass</td>
</tr>
<tr>
<td>7. PDFiD successfully disables JavaScript and auto launch code.</td>
<td><code>pdfid.py –d [filename]</code></td>
<td>manual, benign, malicious</td>
<td>A file is created named [filename].disarmed.pdf in which JavaScript and auto launch is disabled. This is detailed in Appendix A.</td>
</tr>
<tr>
<td>8. PDFiD successfully takes a URL as input.</td>
<td><code>pdfid.py [URL]</code></td>
<td><a href="https://ncjrs.gov/pdffiles1/nij/236224.pdf">https://ncjrs.gov/pdffiles1/nij/236224.pdf</a></td>
<td>This command worked when the URL was prepended with http://.</td>
</tr>
<tr>
<td>9. PDFiD does not throw a statistically significant false positive on 5, 10, and 50 MB of random data.</td>
<td><code>pdfid.py –f [filename]</code></td>
<td>random data file [see Appendix A for details]</td>
<td>Appendix A describes the method used to test this assumption. The 50 MB file threw nine false positives, for a total of 27 bytes, a false positive percentage of about .00002%.</td>
</tr>
</tbody>
</table>

*Note: Each command was executed using the Windows Command Line. This table shows each test assumption, whether or not the assumption passed its test, and details relating to how it was tested and evaluated. Additional information can be found in Appendix A.*

The test assumption regarding entropy requires further explanation. Hands on experience with tools, especially tools that lack documentation like PDFiD, can give the tester a deeper understanding of the tool being reviewed, causing more relevant assumption criteria to become apparent. A final assumption, in addition to what is shown in Table 2, was tested: **PDFiD accurately detects malicious PDFs.** This is a difficult assumption to test for reasons that are expanded upon in the Recommendations for Further Research section of this paper. To prepare
for this test, two more malicious PDF files were acquired from Utica College’s ECJS website, which will be referred to as *mal2* and *mal3*.

In order to test the assumption that *PDFiD* accurately detects malicious PDFs, the command `pdfid -e [filename]` was executed on all three malicious files. None of the three known to be malicious files contained data after the final PDF file footer, and none contained the suspicious string `/OpenAction`. The entropy inside and outside of streams for the three malicious PDFs were 5.9/5.1, 7.9/5.1 and 7.8/5.2, respectively. None of these entropy measurements are consistent with what Didier Stevens described on his blog as being suspicious (Stevens, 2009). However, his blog only addressed PDF files with malicious code appended to the file, a method which none of the malicious PDFs acquired from Utica’s ECJS site made use of. Therefore, this assumption cannot be fully tested until a thorough review of modern techniques of spreading malware via PDF files, a limitation that is acknowledged in the Recommendations for Further Research section of this paper.

**Stress test the tool.** *PDFiD* successfully ran on a directory filled with 500 MB of various files. *PDFiD* successfully took a 500 MB file of random data as input, but its execution lasted longer than 20 minutes, and resulted in 61, or 183 total bytes worth of false positives. *PDFiD* was not tested beyond 500 MB. *PDFiD* failed to measure the entropy of any amount of random data as input, and threw an error that the program was attempting to and incapable of dividing by zero. *PDFiD* threw the same divide by zero error when it took an encrypted file containing a few benign word documents as input.

**Testing Conclusions, Recommendations and Best Practices**

*PDFiD* is a functional and a helpful tool for malware analysts and forensic examiners. *PDFiD* can look through PDF files and find instances of suspicious strings, even when the strings
are obfuscated. In this way, PDFiD is helpful when trying to separate benign PDFs and potentially malicious PDFs.

The following is a fictional example intended to illustrate PDFiD’s proper use. Assume an examiner was given the image file of a hard drive that may or may not have been infected with malware and that the examiner is using a Windows machine for examination. Further assume that a search on the hard drive only found PDF files in the C:\Downloads and the C:\Documents folders, and that the owner of the hard drive claimed to have downloaded and opened a PDF file from an unfamiliar e-mail address.

The examiner should begin by using the –s flag two times, once for each directory: pdfid.py –s C:\Downloads and pdfid.py –s C:\Documents. Doing so will either run PDFiD on the file, provided it is a PDF, or return an error for files that lack the PDF extension or proper headers and footers; a sample output can be seen in Figure 1 in the Analysis section in this paper. In light of traditional methods that malware spreads in PDF files as described in the Analysis section, the examiner should mark each file that contains any or all of the following strings: /Javascript, /AA, /JS, and /OpenAction. Any files that were found to have name obfuscation should be marked because obfuscation indicates an author’s attempt to hide their code. The examiner should run the command pdfid.py –e on each of the marked files to give the reviewer data about the file’s entropy. Files with multiple PDF footers (%%EOF) or files with a significant amount of bytes following the final PDF footer should be marked, as this is a common method used to spread malware via PDFs. The files and their associated marks should be tallied. The more marks a PDF accumulates, the more suspicious it is, and the more cautiously it should be approached.

The goal of using PDFiD this way is to reduce a large set of PDF files into a manageable
set, and to separate the clearly benign PDFs from the potentially malicious PDFs. PDFiD attempts to detect what a given PDF file is capable of executing. The examiner could then use a tool like Didier Stevens’ PDF parser to look at what the tool actually executes (Stevens, 2008). Employing these methods could guide an entire investigation: PDFiD is used to detect the potential malady of a PDF file by recognizing its capabilities. PDF parser is then used to understand how the PDF’s capabilities are actually executed, which could lead the examiner to further investigate specific network activity, shell commands, log events, or registry entries.

**Discussion of the Findings**

This research identified that closed source forensic tools have certain advantages in court. The research also found that with proper documentation, review, and rigor, open source tools can enjoy these same advantages. Specifically, the U.S. v. Springfield case highlighted two areas where AccessData’s FTK, and closed source tools in general, have proven advantages over open source tools. The first advantage is in documentation and training. Open source tools are often side projects, passion projects, or projects that lack traditional corporate incentives like profitability. Therefore, authors of OSS often lack the resources and the incentives to create sufficient documentation and training programs. The U.S. v. Springfield examiner’s evidence was admissible because the examiner had a sufficient understanding of how the tool could be used and what it is capabilities were; an understanding gained due to prior use and training.

The second advantage that closed source tools frequently have is objective training, certificates backing their legitimacy, and trusted third parties. In the case of FTK, AccessData publicizes and administers its own series of FTK-specific certifications (ACE Frequently Asked Questions, 2013). Experts and professionals who work for AccessData and who have access to FTK’s proprietary code have previously testified to the legitimacy of FTK’s functionality.
Analogous third parties or a positive history in court do not exist for open source tools.

**Lack of Foundational Research**

In 2004, the NIJ published a report that details the best practices when conducting examinations of digital evidence (Forensic Examination of Digital Evidence: A Guide for Law Enforcement, 2004). The report articulates the best practices in many high level steps that constitute a sound investigation. The report recommends that evidence is acquired using “stand-alone duplication software” and/or a “forensic analysis software suite”, but doesn’t indicate what factors contribute to an overall level of trust in a given tool or software suite (Chapter 3. Evidence Acquisition, 2004). During the course of this research, no formalized, generally accepted methodology with a positive history in court for testing open source tools was found. Personal communication with Erin Kenneally (2013), author of the 2001 Gatekeeping Out Of The Box paper quoted in the Literature Review, affirmed her belief that open source forensic tools were valuable and acknowledged the lack of legal progress, and she speculated that “much of what I wrote has not been challenged”.

One way this research dealt with this lack of foundational research was to take NIST’s CFTT reports and methods used to review and validate closed source tools, and then to repurpose and apply them to open source tools. Similarly, in 2001, NIST published a whitepaper claiming that its approach to testing digital forensics tools was guided by and was itself partially repurposed from the ISO standard “ISO/IEC 17025, General Requirements for the Competence of Testing and Calibration Laboratories” (General Test Methodology for Computer Forensic Tools, 2. Background, 2001). Another way this paper addressed the lack of research was to work backwards from the Daubert standard and the Federal Rule of Evidence 702 in an attempt to satisfy them in advance (Hutchinson, 2012).
Proving Daubert Compliance

There is no panacea that will prove a given tool or its resultant output is fully compliant with the Daubert standard. Nor is the Daubert standard of evidence a checklist of stipulations that must be met; rather, it is a set of guidelines that a judge uses to determine whether or not evidence is legitimate and admissible. While Brian Carrier and Erin Kenneally’s views echo each other in unison regarding the inherent admissibility and legitimacy of OSS, those views remain to be realized in court. Kenneally and Carrier believe there is value in using and reviewing OSS forensic tools, but neither of them point to a methodology or a trusted entity actually conducting OSS reviews.

The only centralized entities conducting forensic tool reviews were NIST’s CFTT and the FBI/Secret Service’s SWGDE. Both of these entities focus on closed source tools and the legitimacy of the tool’s output. Their reviews rarely examined OSS, and don’t make use of many review techniques that would satisfy Daubert criteria such as sandboxing, code review, stress testing, and error rate measuring.

The U.S. v. Springfield case that Susan Brenner described demonstrates clearly how unimpeachable a tool’s output can be if the tool is backed up by proper documentation, the examiner is trained, and the tool has a history in court. This decision could be viewed as a motivating factor for conducting this research. Sufficient review can render examiners immune to each attack levied at the examiner in U.S. v. Springfield, advantages that have yet to be realized by the CFTT or SWGDE. In addition, a formal method of addressing erroneous output and peer review was articulated, directly addressing additional Daubert criteria, preventing the need to parry shortcomings and inconsistencies as they were in the Springfield case. This was done to improve upon the CFTT and SWGDE’s models for tool review, and to make OSVAT’s
method of review more relevant for OSS.

**OSVAT as a trusted third party.** This research found no trusted third party such as NIST that reviews and certifies open source forensic tools. To address this shortcoming, it is proposed that the OSVAT website be created, and that reviews of OSS be conducted along the lines described in the Methodology section. Specifically, the OSVAT website would be a neutral third party establishing OSS tools compliance with Daubert by demonstrating peer review, general acceptance, and the reliability of principles and methods.

Once the reliability of a tool has been confirmed and documented, OSVAT’s findings and conclusions can be used to train examiners. By reading and understanding this paper, the reader should have a feel for what PDFiD does and how it works. Reading the Analysis section of this paper and using it as a basis for conducting a mock investigation using PDFiD should equip the reader with enough knowledge and expertise with the tool that would allow them to testify to their knowledge, hopefully gaining the advantages afforded by the forensic expert in the U.S. v. Springfield case.

**Recommendations for Further Research**

**Ensuring PDFiD’s Daubert Compliance**

Due to time constraints, certain steps in the review process described in the methodology section weren’t addressed in the analysis section. Classes and functions were not thoroughly analyzed, and stress testing wasn’t fully addressed. However, these are important parts of the process and should be conducted through future research to ensure a complete and thorough validation and testing of the tool. Future research should examine PDFiD closely in these areas to further lend credibility to the overall review and the legitimacy of the tool’s output.

This research did not fully address the subject of entropy, its relevance or effectiveness in
identifying malicious PDFs, or PDFiD’s accuracy when measuring entropy. Similarly, this research didn’t address common or novel attack vectors used by malicious PDFs, or advanced methods of code obfuscation. Further research should seek to answer the following questions:

Does PDFiD accurately measure a PDF’s entropy?
Are certain levels of entropy an indication of malicious code?
What would constitute a complete list of code obfuscation techniques?
What code obfuscation techniques can PDFiD detect?
What are some common or novel ways malicious PDFs have been known to spread in the wild, and would PDFiD detect any unique properties in those files?

Securing Closed Source’s Inherent Advantages

Documentation. One recommendation for further research is to explore how to translate a reasonably technical review, like this one, into a guide that relative laypeople can use to understand how a tool works. After a guide is created, an associated curriculum could be developed and attached to it that would be used to expand the layperson’s understanding from how the tool works to how the tool is used. The end goal is to make an examiner capable of testifying to what the tool does and how it accomplishes its tasks in a manner similar to the examiner in U.S. v. Springfield. Research could address how such a third party could be owned, operated, and funded.

Gaining trust. Further research into the real world legal requirements of OSS-generated evidence’s admissibility in court is needed. Much of the discussion in this paper has been theoretical, and further research should explore what happens when an OSS tool actually is accepted or rejected and why. Additionally, research should be done to look into the efficacy of an OSS Scopes Monkey Trial type legal strategy. In 1925, the ACLU was looking to challenge
some of Tennessee’s anti-evolution laws on free speech grounds. The ACLU found a science teacher called John Scopes working in Tennessee who was willing to teach evolution in school and face indictment, if it came to that (Adams, 2005). In the same way, research should look into the possibility of finding someone willing to conduct a forensic examination using OSS tools that is likely to make its way into court in the attempt to give a tool and/or OSS tools in general a positive foundation in court.

**Daubert compliance.** Future research should look at the problem of the variability and reliability of a tool’s output. What criteria would need to be satisfied to prove that a tool’s output is legitimate? At what point is a tool too unreliable to use for investigations? For example, during this paper’s testing, PDFiD took 50 MB as input and generated nine false positives. To clarify, those nine false positives spanned only 27 bytes worth of data, meaning the tool only stumbled on roughly .00002% of the data it was presented. However, this test was conducted only once. How many times should a tool be tested normally or stress tested until the legitimacy of its output can be proven beyond reasonable doubt?

**Conclusion**

It is hoped that this research will help forensic examiners use PDFiD, while providing the evidence PDFiD generates a stronger foundation in court. It is further hoped that this research expands upon the findings of Brian Carrier and Erin Kenneally, further adding legitimacy to the idea that OSS can produce court admissible evidence while providing a template detailing how it can be accomplished. All things being equal, using transparent methods to generate evidence engenders evidence that is more reliable than evidence generated by hidden or obfuscated means. If this paper’s findings gain traction and are expanded upon, forensic examinations will be conducted with increasingly reliable tools and methods for little or no extra cost. Widely
accepted forensic OSS could ensure cheaper and more accurate legal proceedings, which is a net gain for everyone.
References


Hofman, C. (2013, August 2). *Sandboxes Explained: How They’re Already Protecting You and How to


NIST. (2013, May). *Test Results for Digital Data Acquisition Tool: FTK Imager CLI 2.9.0_Debian.*


PDFiD (Version 0.1.2) [Computer software]. Didier Stevens, no Copyright.


stopped-rewarding-projects.html


Appendix A

PDFiD’s testing methodology and confirmation

PDFiD’s command and its resultant output.

Figure 1. The first test was to check whether each string was accurately detected and counted by PDFiD. The results of this test were confirmed using Notepad ++.

Figure 2. Notepad ++ was used to count each instances of the string /JavaScript. The first test is confirmed, as shown.
The string of number signs preceding the string “8 0 R” that follows the backslash on line 8 is an obfuscation technique. When the PDF file is run, the hexadecimal numbers are translated into the string “OpenAction” which the tool detected, and correctly identified the string as being obfuscated.

The original file, mala.pdf, was ‘disarmed’ by PDFiD, resulting in the file mala.disarmed.pdf. The three names on lines 8-10 were originally /OpenAction /JavaScript and /JS, respectively. By swapping lowercase for uppercase, and upper for lower, the Open Action name and the JavaScript names lose their functionality.

In order to check the false positives that PDFiD occasionally generates, the following commands were run in the virtual Ubuntu OS described in the Analysis section:

- dd if=/dev/urandom of=random5 bs=1000000 count=5
  - python pdfid.py –f random5
    - Detected 0 false positives.
- dd if=/dev/urandom of=random10 bs=1000000 count=10
- python pdfid.py –f random10
  - Detected 1 false positive of the string ‘/AA’.
- dd if=/dev/urandom of=random50 bs=1000000 count=50
  - python pdfid.py –f random50
  - Detected 9 false positives: one instance of ‘obj’ and four instances each of ‘/JS/’ and ‘/AA’.

![Regular PDF data vs. Appended data](http://blog.didierstevens.com/2009/05/14/malformed-pdf-documents/)

*Figure 5. This image was taken from [http://blog.didierstevens.com/2009/05/14/malformed-pdf-documents/](http://blog.didierstevens.com/2009/05/14/malformed-pdf-documents/) and shows the difference between normal regular PDF data and data of high entropy, appended data that is potentially malicious.*
## Appendix B

### Sandboxing Tool Recommendations

Table 1

<table>
<thead>
<tr>
<th>Tool</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anubis</td>
<td><a href="http://Anubis.iselab.org/">http://Anubis.iselab.org/</a></td>
</tr>
<tr>
<td>Joebox</td>
<td><a href="http://www.joesecurity.org/">http://www.joesecurity.org/</a></td>
</tr>
<tr>
<td>Tripwire</td>
<td><a href="http://www.tripwire.org/">http://www.tripwire.org/</a></td>
</tr>
<tr>
<td>Sandboxie</td>
<td><a href="http://www.sandboxie.com/">http://www.sandboxie.com/</a></td>
</tr>
<tr>
<td>Zero Wine Tryout</td>
<td><a href="http://zerowine-tryout.sourceforge.net/">http://zerowine-tryout.sourceforge.net/</a></td>
</tr>
<tr>
<td>CWSandbox</td>
<td><a href="http://mwanalysis.org/">http://mwanalysis.org/</a></td>
</tr>
<tr>
<td>ThreatExpert</td>
<td><a href="http://www.threatexpert.com/">http://www.threatexpert.com/</a></td>
</tr>
</tbody>
</table>